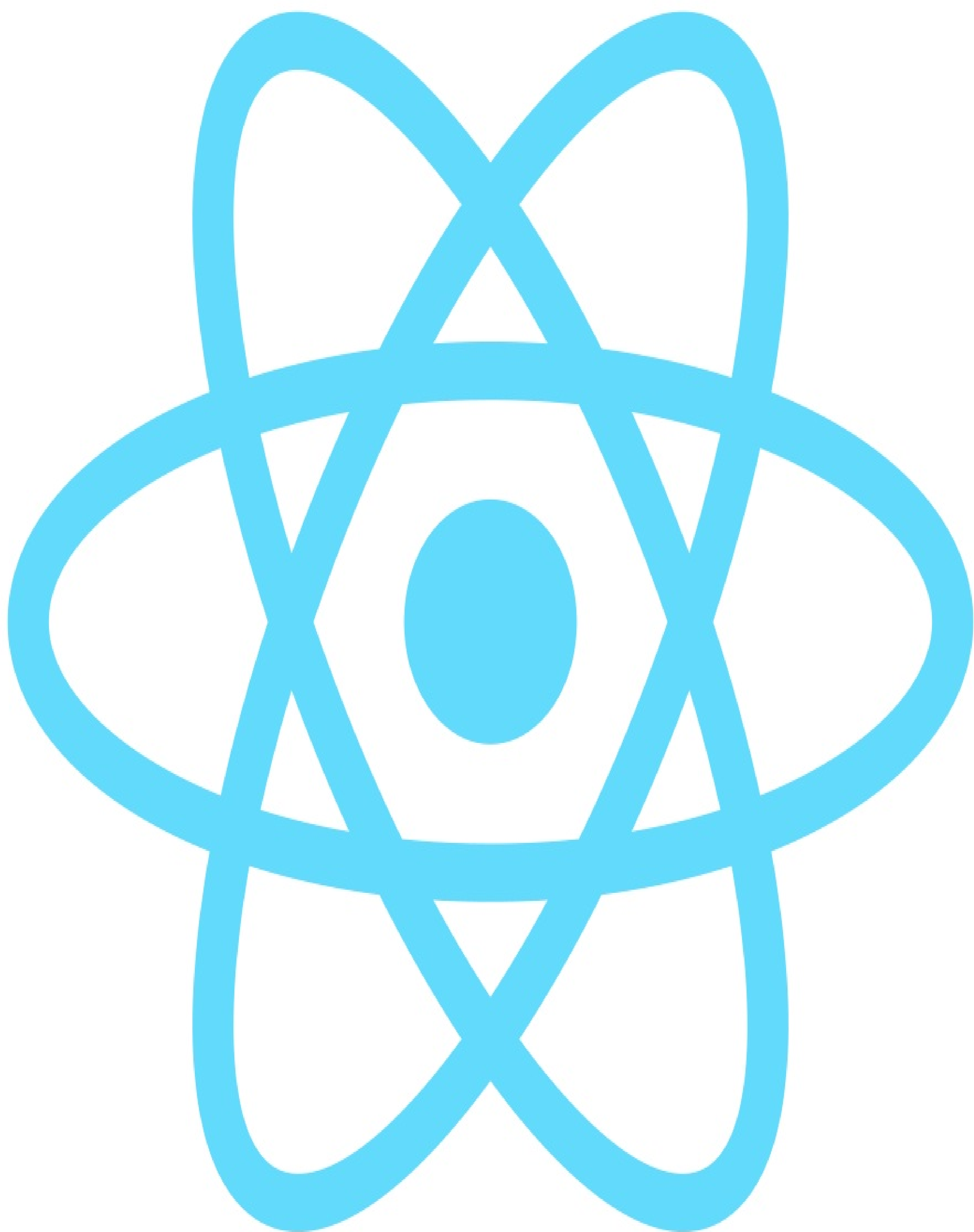


Table of Contents

React Enlightenment

Written by Cody Lindley sponsored by — Frontend Masters

Learn React in the terse cookbook style found with previous Enlightenment titles (i.e., jQuery Enlightenment, JavaScript Enlightenment, DOM Enlightenment)

Read Online At:

- reactenlightenment.com

download a .pdf, .epub, or .mobi file from:

- https://www.gitbook.com/book/frontendmasters/react-enlightenment/details

contribute content, suggestions, and fixes on github:

- https://github.com/FrontendMasters/react-enlightenment



What is React?

React is a JavaScript tool that makes it easy to reason about, construct, and maintain stateless and stateful user interfaces. It provides the means to declaratively define and divide a UI into UI components (a.k.a., React components) using HTML-like nodes called React nodes. React nodes eventually get transformed into a format for UI rendering (e.g., HTML/DOM, canvas, svg, etc.).

I could ramble on trying to express in words what React is, but I think it best to just show you. What follows is a whirlwind tour of React and React components from thirty thousand feet. Don't try and figure out all the details yet as I describe React in this section. The entire book is meant to unwrap the details showcased in the following overview. Just follow along grabbing a hold of the big concepts for now.

Using React to create UI components similar to a `<select>`

Below is an HTML `<select>` element that encapsulates child HTML `<option>` elements. Hopefully the creation and functionality of an HTML `<select>` is already familiar.

> source code

When a browser parses the above tree of elements it will produce a UI containing a textual list of items that can be selected. Click on the "Result" tab in the above JSFiddle, to see what the browser produces.

The browser, the DOM, and the shadow DOM are working together behind the scenes to turn the `<select>` HTML into a UI component. Note that the `<select>` component allows the user to make a selection thus storing the state of that selection (i.e., click on "Volvo", and you have selected it instead of "Mercedes").

Using React you can create a custom `<select>` by using React nodes to make a React component that eventually will result in HTML elements in an HTML DOM.

Let's create our own `<select>`-like UI component using React.

Defining a React component

Below I am creating a React component by invoking the `React.createClass` function in order to create a `MySelect` React component.

As you can see, the `MySelect` component is made up of some styles and an empty React `<div>` node element.

```
var MySelect = React.createClass({ //define MySelect component
    render: function(){
        var mySelectStyle = {
            border: '1px solid #999',
            display: 'inline-block',
            padding: '5px'
        };
        // using {} to reference a JS variable inside of JSX
        return <div style={mySelectStyle}></div>; //react div element, via JSX
    }
});
```

That `<div>` is an HTML-like tag, yes in JavaScript, called JSX. JSX is an optional custom JavaScript syntax used by React to express React nodes that can map to real HTML elements, custom elements, and text nodes. React nodes, defined using JSX should not be considered a one to one match to HTML elements. There are differences and some gotchas.

JSX syntax must be transformed from JSX to real JavaScript in order to be parsed by ES5 JS engines. The above code, if not transformed would of course cause a JavaScript error.

The official tool used to transform JSX to actual JavaScript code is called Babel.

After Babel (or something similar) transforms the JSX `<div>` in the above code into real JavaScript, it will look like this:

```
return React.createElement('div', { style: mySelectStyle });
```

instead of this:

```
return <div style={mySelectStyle}></div>;
```

For now, just keep in mind that when you write HTML-like tags in React code, eventually it must be transformed into real JavaScript, along with any ES6 syntax.

The `<MySelect>` component at this point consist of an empty React `<div>` node element. That's a rather trivial component, so let's change it.

I'm going to define another component called `<MyOption>` and then use the `<MyOption>` component within the `<MySelect>` component (a.k.a., composition).

Examine the updated JavaScript code below which defines both the `<MySelect>` and `<MyOption>` React components.

```
var MySelect = React.createClass({
    render: function(){
        var mySelectStyle = {
            border: '1px solid #999',
            display: 'inline-block',
            padding: '5px'
        };
        return ( //react div element, via JSX, containing <MyOption> component
            <div style={mySelectStyle}>
                <MyOption value="Volvo"></MyOption>
                <MyOption value="Saab"></MyOption>
                <MyOption value="Mercedes"></MyOption>
                <MyOption value="Audi"></MyOption>
            </div>
        );
    }
});

var MyOption = React.createClass({  //define MyOption component
    render: function(){
        return <div>{this.props.value}</div>; //react div element, via JSX
    }
});
```

You should note how the `<MyOption>` component is used inside of the `<MySelect>` component and that both are created using JSX.

Passing Component Options Using React attributes/props

Notice that the `<MyOption>` component is made up of one `<div>` containing the expression `{this.props.value}`. The `{}` brackets indicate to JSX that a JavaScript expression is being used. In other words, inside of `{}` you can write JavaScript.

The `{}` brackets are used to gain access (i.e., `this.props.value`) to the properties or attributes passed by the `<MyOption>` component. In other words, when the `<MyOption>` component is rendered the `value` option, passed using an HTML-like attribute (i.e., `value="Volvo"`), will be placed into the `<div>`. These HTML looking attributes are considered React attributes/props. React uses them to pass stateless/immutable options into components. In this case we are simply passing the `value` prop to the `<MyOption>` component. Not unlike how an argument is passed to a JavaScript function. And in fact that is exactly what is going on behind the JSX.

Rendering a component to the Virtual DOM, then HTML DOM

At this point our JavaScript only defines two React Components. We have yet to actually render these components to the Virtual DOM and thus to the HTML DOM.

Before we do that I'd like to mention that up to this point all we have done is define two React components using JavaScript. In theory, the JavaScript we have so far is just the definition of a UI component. It doesn't strictly have to go into a DOM, or even a Virtual DOM. This same definition, in theory, could also be used to render this component to a native mobile platform or an HTML canvas. But we're not going to do that, even though we could. Just be aware that React is a pattern for organizing a UI that can transcend the DOM, front-end applications, and even the web platform.

Let's now render the `<MySelect>` component to the virtual DOM which in turn will render it to the actual DOM inside of an HTML page.

In the JavaScript below notice I added a call to the `ReactDOM.render()` function on the last line. Here I am passing the `ReactDOM.render()` function the component we want to render (i.e., `<MySelect>`) and a reference to the HTML element already in the HTML DOM (i.e., `<div id="app"></div>`) where I want to render my React `<MySelect>` component.

Click on the "Result" tab and you will see our custom React `<MySelect>` component rendered to the HTML DOM.

> source code

Note that all I did was tell React where to start rendering components and which component to start with. React will then render any children components (i.e., `<MyOption>`) contained within the starting component (i.e., `<MySelect>`).

Hold up, you might be thinking. We haven't actually created a `<select>` at all. All we have done is create a static/stateless list of text strings. We'll fix that next.

Before I move on I want to point out that no implicit DOM interactions were written to get the `<MySelect>` component into the real DOM. In other words, no jQuery code was invoked during the creation of this component. The dealings with the actual DOM have all been abstracted by the React virtual DOM. In fact, when using React what you are doing is describing a virtual DOM that React takes and uses to create a real DOM for you.

Using React state

In order for our `<MySelect>` component to mimic a native `<select>` element we are going to have to add state. After all what good is a custom `<select>` element if it can't keep the state of the selection.

State typically gets involved when a component contains snapshots of information. In regards to our custom `<MyOption>` component, its state is the currently selected text or the fact that no text is selected at all. Note that state will typically involve user events (i.e., mouse, keyboard, clipboard, etc.) or network events (i.e., AJAX) and its value is used to determine when the UI needs to be re-rendered (i.e., when value changes re-render).

State is typically found on the top most component which makes up a UI component. Using the React `getInitialState()` function we can set the default state of our component to `false` (i.e., nothing selected) by returning a state object when `getInitialState` is invoked (i.e., `return {selected: false};`). The `getInitialState` lifecycle method gets invoked once before the component is mounted. The return value will be used as the initial value of `this.state`.

I've updated the code below accordingly to add state to the component. As I am making updates to the code, make sure you read the JavaScript comments which call attention to the changes in the code.

```
var MySelect = React.createClass({
    getInitialState: function(){ //add selected, default state
        return {selected: false}; //this.state.selected = false;
    },
    render: function(){
        var mySelectStyle = {
            border: '1px solid #999',
            display: 'inline-block',
            padding: '5px'
        };
        return (
            <div style={mySelectStyle}>
                <MyOption value="Volvo"></MyOption>
                <MyOption value="Saab"></MyOption>
                <MyOption value="Mercedes"></MyOption>
                <MyOption value="Audi"></MyOption>
            </div>
        );
    }
});

var MyOption = React.createClass({
    render: function(){
        return <div>{this.props.value}</div>;
    }
});

ReactDOM.render(<MySelect />, document.getElementById('app'));
```

With the default state set, next we'll add a callback function called `select` that gets called when a user clicks on an option. Inside of this function we get the text of the option that was selected (via the `event` parameter) and use that to determine how to `setState` on the current component. Notice that I am using `event` details passed to the `select` callback. This pattern should look familiar if you've had any experience with jQuery.

```
var MySelect = React.createClass({
    getInitialState: function(){
        return {selected: false};
    },
    select:function(event){// added select function
        if(event.target.textContent === this.state.selected){//remove selection
            this.setState({selected: false}); //update state
        }else{//add selection
            this.setState({selected: event.target.textContent}); //update state
        }
    },
    render: function(){
        var mySelectStyle = {
            border: '1px solid #999',
            display: 'inline-block',
            padding: '5px'
        };
        return (
            <div style={mySelectStyle}>
                <MyOption value="Volvo"></MyOption>
                <MyOption value="Saab"></MyOption>
                <MyOption value="Mercedes"></MyOption>
                <MyOption value="Audi"></MyOption>
            </div>
        );
    }
});

var MyOption = React.createClass({
    render: function(){
        return <div>{this.props.value}</div>;
    }
});

ReactDOM.render(<MySelect />, document.getElementById('app'));
```

In order for our `<MyOption>` components to gain access to the `select` function we'll have to pass a reference to it, via props, from the `<MySelect>` component to the `<MyOption>` component. To do this we add `select={this.select}` to the `<MyOption>` components.

With that in place we can add `onClick={this.props.select}` to the `<MyOption>` components. Hopefully it's obvious that all we have done is wired up a `click` event that will call the `select` function. React takes care of wiring up the real click handler in the real DOM for you.

```
var MySelect = React.createClass({
    getInitialState: function(){
        return {selected: false};
    },
    select:function(event){
        if(event.target.textContent === this.state.selected){
            this.setState({selected: false});
        }else{
            this.setState({selected: event.target.textContent});
        }
    },
    render: function(){
        var mySelectStyle = {
            border: '1px solid #999',
            display: 'inline-block',
            padding: '5px'
        };
        return (//pass reference, using props, to select callback to <MyOption>
            <div style={mySelectStyle}>
                <MyOption select={this.select} value="Volvo"></MyOption>
                <MyOption select={this.select} value="Saab"></MyOption>
                <MyOption select={this.select} value="Mercedes"></MyOption>
                <MyOption select={this.select} value="Audi"></MyOption>
            </div>
        );
    }
});

var MyOption = React.createClass({
    render: function(){//add event handler that will invoke select callback
        return <div onClick={this.props.select}>{this.props.value}</div>;
    }
});

ReactDOM.render(<MySelect />, document.getElementById('app'));
```

By doing all this we can now set the state by clicking on one of the options. In other words, when you click on an option the `select` function will now run and set the state of the `MySelect` component. However, the user of the component has no idea this is being done because all we have done is update our code so that state is managed by the component. At this point we have no feedback visually that anything is selected. Let's fix that.

The next thing we will need to do is pass the current state down to the `<MyOption>` component so that it can respond visually to the state of the component.

Using props, again, we will pass the `selected` state from the `<MySelect>` component down to the `<MyOption>` component by placing the property `state={this.state.selected}` on all of the `<MyOption>` components. Now that we know the state (i.e., `this.props.state`) and the current value (i.e., `this.props.value`) of the option we can verify if the state matches the

value in a given `<MyOption>` component . If it does, we then know that this option should be selected. We do this by writing a simple `if` statement which adds a styled selected state (i.e., `selectedStyle`) to the JSX `<div>` if the state matches the value of the current option. Otherwise, we return a React element with `unSelectedStyle` styles.

> source code

Make sure you click on the "Result" tab above and use the custom React select component to verify the new functioning.

While our React UI select component is not as pretty or feature complete as you might hope, I think you can see still where all this is going. React is a tool that can help you reason about, construct, and maintain stateless and stateful UI components, in a structure tree (i.e., a tree of components).

Before moving on to the role of the virtual DOM I do want to stress that you don't have to use JSX and Babel. You can always bypass these tools and just write straight JavaScript. Below, I'm showing the final state of the code after the JSX has been transformed by Babel. If you choose not to use JSX, then you'll have to write the following code yourself instead of the code I've written throughout this section.

```
var MySelect = React.createClass({
  displayName: 'MySelect',

  getInitialState: function getInitialState() {
    return { selected: false };
  },
  select: function select(event) {
    if (event.target.textContent === this.state.selected) {
      this.setState({ selected: false });
    } else {
      this.setState({ selected: event.target.textContent });
    }
  },
  render: function render() {
    var mySelectStyle = {
      border: '1px solid #999',
      display: 'inline-block',
      padding: '5px'
    };
    return React.createElement(
      'div',
      { style: mySelectStyle },
      React.createElement(MyOption, { state: this.state.selected, select: this.select,
 value: 'Volvo' }),
      React.createElement(MyOption, { state: this.state.selected, select: this.select,
 value: 'Saab' }),
      React.createElement(MyOption, { state: this.state.selected, select: this.select,
 value: 'Mercedes' }),
```

```
      React.createElement(MyOption, { state: this.state.selected, select: this.select,
 value: 'Audi' })
    );
  }
});

var MyOption = React.createClass({
  displayName: 'MyOption',

  render: function render() {
    var selectedStyle = { backgroundColor: 'red', color: '#fff', cursor: 'pointer' };
    var unSelectedStyle = { cursor: 'pointer' };
    if (this.props.value === this.props.state) {
      return React.createElement(
        'div',
        { style: selectedStyle, onClick: this.props.select },
        this.props.value
      );
    } else {
      return React.createElement(
        'div',
        { style: unSelectedStyle, onClick: this.props.select },
        this.props.value
      );
    }
  }
});

ReactDOM.render(React.createElement(MySelect, null), document.getElementById('app'));
```

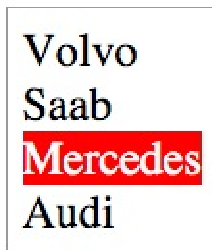
Understanding the role of the Virtual DOM

I'm going to end this whirlwind tour where most people typically start talking about React. I'll finish off this React overview by talking about the merits of the React virtual DOM.

Hopefully you notice the only interaction with the real DOM we had during the creation of our custom select UI is when we told the `ReactDOM.render()` function where to render our UI component in the HTML page (i.e., render it to `<div id="app"></div>`). This might just be the only interaction you ever have with the real DOM when building out a React application from a tree of React components. And herein lies much of the value of React. By using React, you really don't ever have to think about the DOM like you once did when you were writing jQuery code. React replaces jQuery, as a complete DOM abstraction, by removing most, if not all, implicit DOM interactions from your code. Of course, that's not the only benefit, or even the best benefit.

Because the DOM has been completely abstracted by the Virtual DOM this allows for a heavy handed performance pattern of updating the real DOM when state is changed. The Virtual DOM keeps track of UI changes based on state and props. It then compares that to the real DOM, and then makes only the minimal changes required to update the UI. In other words, the real DOM is only ever patched with the minimal changes needed when state or props change.

Seeing these performant updates in real time will often clarify any confusion about the performant DOM diffing. Look at the animated image below showcasing the usage (i.e., changing state) of the UI component we created in this chapter.



Notice that as the UI component changes state only the minimally needed changes to the real DOM are occurring. We know that React is doing it's job because the only parts of the real DOM that are actually being updated are the parts with a green outline/background. The entire UI component is not being updated on each state change, only the parts that require a change.

Let me be clear, this isn't a revolutionary concept. You could accomplish the same thing with some carefully crafted and performant minded jQuery code. However, by using React you'll rarely, if ever, have to think about it. The Virtual DOM is doing all the performance work for you. In a sense, this is the best type of jQuery/DOM abstraction possible. One where you don't even have to worry about, or code for, the DOM. It all just happens behinds the scene without you ever implicitly having to interact with the DOM itself.

Now, it might be tempting to leave this overview thinking the value of React is contained in the fact that it almost eliminates the need for something like jQuery. And while the Virtual DOM is certainly a relief when compared to implicit jQuery code, the value of React does not rest alone on the Virtual DOM. The Virtual DOM is just the icing on the cake. Simply stated, the value of React rests upon the fact it provides a simple and maintainable pattern for creating a tree of UI components. Imagine how simple programming a UI could be by defining the entire interface of your application using reusable React components alone.

React Semantics

Before I enlighten you with the mechanics of React I'd first like to define a few terms so as to grease the learning process.

Below I list the most common terms, and their definitions, used when talking about React.

Babel

Babel transforms JavaScript ES* (i.e., JS 2016, 2016, 2017) to ES5. Babel is the tool of choice from the React team for writing future ES* code and transforming JSX to ES5 code.

Babel CLI

Babel comes with a CLI tool, called Babel CLI, that can be used to compile files from the command line.

Component Configuration Options (a.k.a, "Component Specifications")

The configuration arguments passed (as an object) to the `React.createClass()` function resulting in an instance of a React component.

Component Life Cycle Methods

A sub group of component events, semantically separated from the other component configuration options (i.e., `componentWillUnmount`, `componentDidUpdate`, `componentWillUpdate`, `shouldComponentUpdate`, `componentWillReceiveProps`, `componentDidMount`, `componentWillMount`). These methods are executed at specific points in a component's existence.

Document Object Model (a.k.a., DOM)

"The Document Object Model (DOM) is a programming interface for HTML, XML and SVG documents. It provides a structured representation of the document as a tree. The DOM defines methods that allow access to the tree, so that they can change the document structure, style and content. The DOM provides a representation of the document as a structured group of nodes and objects, possessing various properties and methods. Nodes can also have event handlers attached to them, and once an event is triggered, the event handlers get executed. Essentially, it connects web pages to scripts or programming languages." - MSD

ES5

The 5th edition of the ECMAScript standard. The ECMAScript 5.1 edition was finalized in June 2011.

ES6/ES 2015

The 6th edition of the ECMAScript standard. A.k.a, JavaScript 2015 or ECMAScript 2015. The ECMAScript 6th edition was finalized in June 2015.

ECMAScript 2016 (a.k.a, ES7)

The 7th edition of the ECMAScript standard. The ECMAScript 7th edition was finalized in June 2016.

ES*

Used to represent the current version of JavaScript as well as potential future versions that can written today using tools like Babel. When you see "ES*" it more than likely means you'll find uses of ES5, ES6, and ES7 together.

JSX

JSX is an optional XML-like syntax extension to ECMAScript that can be used to define an HTML-like tree structure in a JavaScript file. The JSX expressions in a JavaScript file must be transformed to JavaScript syntax before a JavaScript engine can parse the file. Babel is

typically used and recommended for transforming JSX expressions.

Node.js

Node.js is an open-source, cross-platform runtime environment for writing JavaScript. The runtime environment interprets JavaScript using Google's V8 JavaScript engine.

---

npm

npm is the package manager for JavaScript born from the Node.js community.

React Attributes/Props

In one sense you can think of props as the configuration options for React nodes and in another sense you can think of them as HTML attributes.

Props take on several roles:

1. Props can become HTML attributes. If a prop matches a known HTML attribute then it will be added to the final HTML element in the DOM.
2. Props passed to `createElement()` become values stored in a `prop` object as an instance property of `React.createElement()` instances (i.e., `[INSTANCE].props.[NAME OF PROP]`). Props by and large are used to input values into components.
3. A few special props have side effects (e.g., `key`, `ref`, and `dangerouslySetInnerHTML`)

React

React is an open source JavaScript library for writing declarative, efficient, and flexible user interfaces.

React Component

A React component is created by calling `React.createClass()` (or, `React.Component` if using ES6 classes). This function takes an object of options that is used to configure and create a React component. The most common configuration option is the `render` function which

returns React nodes. Thus, you can think of a React component as an abstraction containing one or more React nodes/components.

React Element Nodes (a.k.a., `ReactElement`)

An HTML or custom HTML element node representation in the Virtual DOM created using `React.createElement();` .

React Nodes

React nodes (i.e., element and text nodes) are the primary object type in React and can be created using `React.createElement('div');` . In other words React nodes are objects that represent DOM nodes and children DOM nodes. They are a light, stateless, immutable, virtual representation of a DOM node.

React Node Factories

A function that generates a React element node with a particular type property.

React Stateless Function Component

When a component is purely a result of props alone, no state, the component can be written as a pure function avoiding the need to create a React component instance.

```
var MyComponent = function(props){
    return <div>Hello {props.name}</div>;
};

ReactDOM.render(<MyComponent name="doug" />, app);
```

React Text Nodes (a.k.a., `ReactText`)

A text node representation in the Virtual DOM created using `React.createElement('div',null,'a text node');` .

Virtual DOM

An in-memory JavaScript tree of React elements/components that is used for efficient re-rendering (i.e., diffing via JavaScript) of the browser DOM.

Webpack

Webpack is a module loader and bundler that takes modules (.js, .css, .txt, etc.) with dependencies and generates static assets representing these modules.

React Setup

This section will discuss setting up an HTML page so that when it is parsed by a web browser, at runtime, the browser can transform JSX expressions and correctly run React code.

Using `react.js` and `react-dom.js` in an HTML Page

The `react.js` file is the core file needed to create React elements and write react components. When you intend to render your components in an HTML document (i.e., the DOM) you'll also need the `react-dom.js` file. The `react-dom.js` file is dependent on the `react.js` file and must be included after first including the `react.js` file.

An example of an HTML document properly including React is shown below.

```
<!DOCTYPE html>
<html>
  <head>
    <script src="https://fb.me/react-15.2.0.js"></script>
    <script src="https://fb.me/react-dom-15.2.0.js"></script>
  </head>
<body>
</body>
</html>
```

With the `react.js` file and `react-dom.js` file loaded into an HTML page it is then possible to create React nodes/components and then render them to the DOM. In the HTML below a `HelloMessage` React component is created containing a React `<div>` node that is rendered to the DOM inside of the `<div id="app"></div>` HTML element.

```
<!DOCTYPE html>
<html>
    <head>
        <script src="https://fb.me/react-15.2.0.js"></script>
        <script src="https://fb.me/react-dom-15.2.0.js"></script>
    </head>
<body>
    <div id="app"></div>
    <script>
        var HelloMessage = React.createClass({
            displayName: 'HelloMessage',
            render: function render() {
                return React.createElement('div',null,'Hello ',this.props.name);
            }
        });
        ReactDOM.render(React.createElement(HelloMessage,{ name: 'John' }), document.g
etElementById('app'));
    </script>
</body>
</html>
```

This setup is all you need to use React. However this setup does not make use of JSX. The next section will discuss JSX usage.

Notes

- An alternative `react.js` file called `react-with-addons.js` is available containing a collection of utility modules for building React applications. The "addons" file can be used in place of the `react.js` file.
- Don't make the `<body>` element the root node for your React app. Always put a root `<div>` into `<body>`, give it an ID, and render into it. This gives React its own pool to play in without worrying about what else potentially wants to make changes to the children of the `<body>` element.

Using JSX via Babel

The creation of the React `HelloMessage` component and React `<div>` element node in the HTML page below was done using the `React.createClass()` and `React.createElement()` functions. This code should look familiar as it is identical to the HTML from the previous section. This HTML will run without errors in ES5 browsers.

```
<!DOCTYPE html>
<html>
    <head>
        <script src="https://fb.me/react-15.2.0.js"></script>
        <script src="https://fb.me/react-dom-15.2.0.js"></script>
    </head>
<body>
    <div id="app"></div>
    <script>
        var HelloMessage = React.createClass({
            displayName: 'HelloMessage',
            render: function render() {
                return React.createElement('div',null,'Hello ',this.props.name);
            }
        });
        ReactDOM.render(React.createElement(HelloMessage,{ name: 'John' }), document.g
etElementById('app'));
    </script>
</body>
</html>
```

Optionally, by using JSX via Babel, it is possible to simplify the creation of React elements by abstracting the `React.createElement()` JavaScript function calls so they can be written in a more natural HTML-like style and syntax.

Instead of writing the following, which uses `React.createElement()` :

```
return React.createElement('div',null,'Hello ',this.props.name);
```

Using JSX, you can write this:

```
return <div>Hello {this.props.name}</div>;
```

And then Babel will convert it back to the code which uses `React.createElement()` so it can be parsed by a JavaScript engine.

Loosely stated you can consider JSX a form of HTML that you can directly write in JavaScript that requires a transformation step, done by Babel, into ECMAScript 5 code that browsers can run.

More will be said about JSX in the JSX chapter. For now, just realize that JSX is an optional abstraction provided for your convenience when creating React elements and it won't run in ES5 browsers without first being transformed by Babel.

Transforming JSX via Babel in the Browser

Normally, Babel is setup to automatically process your JavaScript files during development using the Babel CLI tool (e.g., via something like webpack). However, it is possible to use Babel directly in the browser by way of a script include. As we are just getting started we'll avoid CLI tools or learning a module loader to focus on learning React. Note that you should never use this browser transformation in a production environment.

The Babel project unfortunately, as of Babel 6, no longer provides the script file needed (i.e., `browser.js`) to transform JSX code to ES5 code in the browser. Thus you will have to use an older version of Babel (i.e., 5.8.23) that provides the needed file (i.e., `browser.js`) for transforming JSX/ES* in the browser.

Using `browser.js` (Babel 5.8.23) to Transform JSX in the Browser

In the HTML file below the React code we have been working with to create the `HelloMessage` component has been updated to use JSX. The transformation of the code is occurring because we have included the `browser.js` Babel file and given the `<script>` element a `type` attribute of `type="text/babel"`.

```
<!DOCTYPE html>
<html>
    <head>
        <script src="https://fb.me/react-15.2.0.js"></script>
        <script src="https://fb.me/react-dom-15.2.0.js"></script>
        <script src="https://cdnjs.cloudflare.com/ajax/libs/babel-core/5.8.23/browser.
min.js"></script>
    </head>
<body>
    <div id="app"></div>
    <script type="text/babel">
        var HelloMessage = React.createClass({
            render: function() {
                return                                   ;
            }
        });

        ReactDOM.render(
    </script>
</body>
</html>
```

While transforming JSX in the browser is convenient and easy to setup, it isn't ideal because the transformation cost is occurring at runtime. This will slow down your web pages and may cause memory issues. Therefore using browser.js is ***not a production solution***.

You should be aware that the code examples used in this book via JSFiddle will be using the `browser.js` (Babel 5.8.23) to transform JSX into ES5 code. In other words, JSFiddle is pretty much doing what you see in the HTML file above when running React code.

Notes

- The Babel tool is a subjective selection from the React team for transforming ES* code and JSX syntax to ES5 code. Learn more about Babel by reading the Babel handbook.
- By using JSX:
    - Less technical people can still understand and modify the required parts. CSS developers and designers will find JSX more familiar than JavaScript alone.
    - You can leverage the full power of JavaScript in HTML and avoid learning or using a templating language. JSX is not a templating solution. It is a declarative syntax used to express a tree structure of UI components.
    - The compiler will find errors in your HTML you might otherwise miss.
    - JSX promotes the idea of inline styles. Which can be a good thing.
- Beware of JSX Gotchas.
- A JSX specification is currently being drafted. It can be used by anyone as an a XML-like syntax extension to ECMAScript without any defined semantics.

Using ES6 & ES* With React

Babel is not part of React. In fact, Babel's purpose isn't even that of a JSX transformer. Babel is a JavaScript compiler first. It takes ES* code and transforms it to run in browsers that don't support ES* code. As of today, Babel mostly takes ES6 and ES7 code and transforms it into ES5 code. When doing these ECMAScript transformations it is trivial to also transform JSX expressions into `React.createElement()` calls. This is what we examined in the previous section.

Given that Babel is the mechanism for transforming JSX, it allows you to write code that will run in future versions of ES*.

In the HTML page below the familiar `HelloMessage` component has been rewritten to take advantage of ES6 classes. Not only is Babel transforming the JSX syntax, it is also transforming ES6 class syntax to ES5 syntax which can then be parsed by ES5 browser engines.

```
<!DOCTYPE html>
<html>
    <head>
        <script src="https://fb.me/react-15.2.0.js"></script>
        <script src="https://fb.me/react-dom-15.2.0.js"></script>
        <script src="https://cdnjs.cloudflare.com/ajax/libs/babel-core/5.8.23/browser.
min.js"></script>
    </head>
<body>
    <div id="app"></div>
    <script type="text/babel">

        class HelloMessage extends React.Component { //notice use of React.Component
            render(){
                return <div>Hello {this.props.name}</div>;
            }
        };

        ReactDOM.render(<HelloMessage name="John" />, document.getElementById('app'));

        /*** PREVIOUSLY ***/
        /* var HelloMessage = React.createClass({
         *    render: function() {
         *        return <div>Hello {this.props.name}</div>;
         *    }
         * });
         *
         * ReactDOM.render(<HelloMessage name="John" />, document.getElementById('app'
));
         */
    </script>
</body>
</html>
```

In the above HTML document Babel is taking in:

```
class HelloMessage extends React.Component {
    render(){
        return <div>Hello {this.props.name}</div>;
    }
};

ReactDOM.render(<HelloMessage name="John" />, document.getElementById('app'));
```

And transforming it to this:

```
"use strict";

var _createClass = (function () { function defineProperties(target, props) { for (var
```

```
i = 0; i < props.length; i++) { var descriptor = props[i]; descriptor.enumerable = des
criptor.enumerable || false; descriptor.configurable = true; if ("value" in descriptor
) descriptor.writable = true; Object.defineProperty(target, descriptor.key, descriptor
); } } return function (Constructor, protoProps, staticProps) { if (protoProps) define
Properties(Constructor.prototype, protoProps); if (staticProps) defineProperties(Const
ructor, staticProps); return Constructor; }; })();

var _get = function get(_x, _x2, _x3) { var _again = true; _function: while (_again) {
var object = _x, property = _x2, receiver = _x3; _again = false; if (object === null)
object = Function.prototype; var desc = Object.getOwnPropertyDescriptor(object, proper
ty); if (desc === undefined) { var parent = Object.getPrototypeOf(object); if (parent
=== null) { return undefined; } else { _x = parent; _x2 = property; _x3 = receiver; _a
gain = true; desc = parent = undefined; continue _function; } } else if ("value" in de
sc) { return desc.value; } else { var getter = desc.get; if (getter === undefined) { r
eturn undefined; } return getter.call(receiver); } } };

function _classCallCheck(instance, Constructor) { if (!(instance instanceof Constructo
r)) { throw new TypeError("Cannot call a class as a function"); } }

function _inherits(subClass, superClass) { if (typeof superClass !== "function" && sup
erClass !== null) { throw new TypeError("Super expression must either be null or a fun
ction, not " + typeof superClass); } subClass.prototype = Object.create(superClass &&
superClass.prototype, { constructor: { value: subClass, enumerable: false, writable: t
rue, configurable: true } }); if (superClass) Object.setPrototypeOf ? Object.setProtot
ypeOf(subClass, superClass) : subClass.__proto__ = superClass; }

var HelloMessage = (function (_React$Component) {
    _inherits(HelloMessage, _React$Component);

    function HelloMessage() {
        _classCallCheck(this, HelloMessage);

        _get(Object.getPrototypeOf(HelloMessage.prototype), "constructor", this).apply(
this, arguments);
    }

    _createClass(HelloMessage, [{
        key: "render",
        value: function render() {
            return React.createElement(
                "div",
                null,
                "Hello ",
                this.props.name
            );
        }
    }]);

    return HelloMessage;
})(React.Component);

;
```

```
ReactDOM.render(React.createElement(HelloMessage, { name: "John" }), document.getEleme
ntById('app'));
```

Most ES6 features with a few caveats can be used when writing JavaScript that is transformed by Babel 5.8.23 (i.e., https://cdnjs.cloudflare.com/ajax/libs/babel-core/5.8.23/browser.js).

Notes

- Obviously one can could still use Babel for it's intended purpose (i.e., compiling newer JavaScript code to older JavaScript code) without using JSX. However, most people using React are taking advantage of Babel for both unsupported ES* features and JSX transforming.
- Learn more about Babel by reading the Babel handbook.

Writing React With JSFiddle

The basic setup that has been described in this chapter can be used online via JSfiddle. JSFiddle uses the same three resources used earlier to make writing React online simple.

- `react-with-addons.js` (the newer version of `react.js`)
- `react-dom.js`
- `browser.js`

Below is an embedded JSFiddle containing the `HelloMessage` component used throughout this chapter. By clicking on the "results" tab you can view the React component rendered to the DOM. To edit the code just click on "edit with JSFiddle".

> source code

Note that the "Babel" tab indicates the JavaScript written into this tab will be transformed by Babel (similar too, if not directly using `browser.js`). The "Resources" tab shows that JSFiddle is pulling in the `react.js` and `react-dom.js` files.

It will be assumed that after reading this chapter that you understand the basic requirements to setup React and Babel via `browser.js` . And that while JSFiddle does not make it obvious, this is the same exact setup being used by JSFiddle to run React code.

JSFiddle will be used throughout the rest of this book to show the results of React code transformed by Babel.

React Nodes

This section will discuss creating React nodes (text or element nodes) using JavaScript.

What Is a React Node?

The primary type or value that is created when using React is known as a React node. A React node is defined as:

> a light, stateless, immutable, virtual representation of a DOM node.

React nodes are not real DOM nodes (e.g., text or element nodes) themselves, but a representation of a potential DOM node. The representation is considered the virtual DOM. In a nutshell, React is used to define a virtual DOM using React nodes, that fuel React components, that can eventually be used to create a real DOM structured or other structures (e.g., React Native).

React nodes can be created using JSX or JavaScript. In this chapter we'll look at creating React nodes using JavaScript alone. No JSX yet. I believe that you should first learn what JSX is concealing in order to understand JSX.

You might be tempted to skip this chapter because you already know that you will be using JSX. I'd suggest reading this chapter so you are aware of what JSX is doing for you. This is likely the most important chapter in the book to Grok.

Creating React Nodes

In most cases developers using React will favor JSX and use it to create React nodes. Despite this, in this chapter we are going to examine how React nodes can be created without JSX, using only JavaScript. The next chapter will discuss creating React nodes using JSX.

Creating React nodes using JavaScript is as simple as calling the `React.createElement(type,props,children)` function and passing it a set of arguments defining an actual DOM node (e.g., type = an html element e.g., `<li>` or custom element e.g., `<my-li>`).

The `React.createElement()` arguments are explained below:

- **type (string | `React.createClass()`):**

  Can be a string which represents an HTML element (or custom HTML element) or React component instance (i.e., an instance of `React.createClass()`)

- **props (null | object):**

  Can be `null` or an object containing attributes/props and values

- **children (null | string | `React.createClass()` | `React.createElement()`):**

  Children can be `null` , a string that gets turned into a text node, an instance of `React.createClass()` or `React.createElement()`

Below I use the `React.createElement()` function to create a virtual DOM representation of a `<li>` element node containing a text node of `one` (a.k.a., React text) and an `id` of `li1` .

```
var reactNodeLi = React.createElement('li', {id:'li1'}, 'one');
```

Notice that the first argument defines the HTML element I want to represent. The second argument defines the attributes/props on the `<li>` . And the third argument defines what the node inside of the element will be. In this case, I am simply placing a child text node (i.e., `'one'`) inside the `<li>` . The last argument that becomes a child of the node being created can be

- A React text node
- A React element node, or
- A React component instance.

At this point all I've done is create a React element node containing a React text node that I have stored into the variable `reactNodeLi`. To create a virtual DOM we have to actually render the React element node to a real DOM. To do this we use the `ReactDOM.render()` function.

```
ReactDOM.render(reactNodeLi, document.getElementById('app'));
```

The above code, loosely stated, invokes the following:

1. Create a virtual DOM constructed from the React element node passed in (`reactNodeLi`)
2. Use the virtual DOM to re-construct a real DOM branch
3. Insert the real DOM branch (i.e., `<li id="li1">one</li>`) into the DOM as a child node of `<div id="app"></div>` .

In other words, the HTML DOM changes from this:

```
<div id="app"></div>
```

to this:

```
<div id="app">
    //note that React added the react data-reactid attribute
    <li id="li1" data-reactid=".0">one</li>
</div>
```

This was a rather simplistic example. Using `React.createElement()` a complex structure can be created as well. For example, below I'm using `React.createElement()` to create a bunch of React nodes representing an HTML unordered list of text words (i.e., `<ul>`).

```
// Create React element <li>'s
var rElmLi1 = React.createElement('li', {id:'li1'}, 'one'),
    rElmLi2 = React.createElement('li', {id:'li2'}, 'two'),
    rElmLi3 = React.createElement('li', {id:'li3'}, 'three');

// Create <ul> React element and add child
// React <li> elements to it
var reactElementUl = React.createElement('ul', {className:'myList'}, rElmLi1, rElmLi2,
 rElmLi3);
```

Before rendering the unordered list to the DOM I think it is worth showing that the above code can be simplified by using the `React.createElement()` in place of variables. This also demonstrates how a hierarchy or DOM branch can be defined using JavaScript.

```
var reactElementUl = React.createElement(
    'ul', {
        className: 'myList'
    },
    React.createElement('li', {id: 'li1'}, 'one'),
    React.createElement('li', {id: 'li2'}, 'two'),
    React.createElement('li', {id: 'li3'}, 'three')
);
```

When the above code is rendered to the DOM the resulting HTML will look like:

```
<ul class="myList" data-reactid=".0">
    <li id="li1" data-reactid=".0.0">one</li>
    <li id="li2" data-reactid=".0.1">two</li>
    <li id="li3" data-reactid=".0.2">three</li>
</ul>
```

You can investigate this yourself using the JSFiddle below:

> source code

It should be obvious that React nodes are just JavaScript objects in a tree that represent real DOM nodes inside of a virtual DOM tree. The virtual DOM is then used to construct an actual DOM branch in an HTML page.

Notes

- The `type` argument passed to `React.createElement(type, props, children)` can be
  - A string indicating a standard HTML element (e.g., `'li'` = `<li></li>`), or
  - A custom element (e.g., `'foo-bar'` = `<foo-bar></foo-bar>`, or a React component instance (i.e., an instance of `React.createClass()`.
- These are the standard HTML elements that React supports (i.e. these elements passed as a string `type` to `createElement()`). They create the associating standard HTML element in the DOM):

```
a abbr address area article aside audio b base bdi bdo big blockquote body br
button canvas caption cite code col colgroup data datalist dd del details dfn
dialog div dl dt em embed fieldset figcaption figure footer form h1 h2 h3 h4 h5
h6 head header hgroup hr html i iframe img input ins kbd keygen label legend li
link main map mark menu menuitem meta meter nav noscript object ol optgroup
option output p param picture pre progress q rp rt ruby s samp script section
select small source span strong style sub summary sup table tbody td textarea
tfoot th thead time title tr track u ul var video wbr
```

Rendering to DOM

React provides the `ReactDOM.render()` function from `react-dom.js` that can be used to render React nodes to the DOM. We've already seen this `render()` function in use in this chapter.

In the code example below, using `ReactDOM.render()`, the `'<li>'` and `'<foo-bar>'` React nodes are rendered to the DOM.

> source code

Once rendered to the DOM, the updated HTML will be:

```
<body>
    <div id="app1"><li class="bar" data-reactid=".0">foo</li></div>
    <div id="app2"><foo-bar classname="bar" children="foo" data-reactid=".1">foo</foo-bar></div>
</body>
```

The `ReactDOM.render()` function is initially how you get the React nodes to the Virtual DOM, then to the HTML DOM.

Notes

- Any DOM nodes inside of the DOM element which you are rendering into will be replaced (i.e., removed).
- `ReactDOM.render()` does not modify the DOM element node in which you are rendering React. However, when rendering React wants complete ownership of the node. You should not add children to or remove children from a node in which React inserts a React node/component.
- Rendering to an HTML DOM is only one option with React, other rendering APi's are available. For example, it is also possible to render to a string (i.e., `ReactDOMServer.renderToString()`) on the server-side.
- Re-rendering to the same DOM element will only update the current child nodes if a change has been made or a new child node has been added.

Defining Node Attributes/Props

The second argument that is passed to `React.createElement(type, props, children)` is an object containing name value properties (a.k.a, props).

Props take on several roles:

1. Props can become HTML attributes. If a prop matches a known HTML attribute then it will be added to the final HTML element in the DOM.
2. Props passed to `createElement()` become values stored in a `prop` object as an instance property of `React.createElement()` instances (i.e., `[INSTANCE].props.[NAME OF PROP]`). Props are normally used to input values into components.
3. A few special props have side effects (e.g., `key`, `ref`, and `dangerouslySetInnerHTML`)

In one sense you can think of props as the configuration options for React nodes and in another sense you can think of them as HTML attributes.

In the code example below I am defining a React `<li>` element node with five props. One is a non-standard HTML attribute (i.e., `foo:'bar'`) and the others are known HTML attributes.

```
var reactNodeLi = React.createElement('li',
    {
        foo: 'bar',
        id: 'li1',
        // Note the use of the JS className property to change the
        // class attribute below
        className: 'blue',
        'data-test': 'test',
        'aria-role' :'listitem',
        // Note use of JS camel-cased CSS property backgroundColor below
        style: {backgroundColor:'red'}
    },
    'text'
);
```

When the above code is rendered to an HTML page the actual HTML created will look like:

```
<li id="li1" data-test="test" class="blue" aria-role="listitem" style="background-colo
r:red;" data-reactid=".0">text</li>
```

What you need to realize is only the following attributes get passed to the real DOM from the Virtual DOM

- Standard HTML attributes,
- Custom data attributes `data-*` , and
- Accessibility attributes `aria-*`

The `foo` attribute/prop does not show up in the real DOM. This non-standard HTML attribute is available as an instance property of the created `li` React node instance. (e.g., `reactNodeLi.props.foo`).

> source code

React attributes/props not only translate to real HTML attributes props, they become configuration values that are passed to React components. This aspect of props will be covered in the React component props chapter. For now simply realize that passing a prop into a React node is different from defining a prop on a component to be used as configuration input within a component.

Notes

- Leaving an attribute/prop blank results in that attribute value becoming true (e.g., `id=""` becomes `id="true"` and `test` becomes `test="true"`)
- If an attribute/prop is duplicated the last one defined wins.
- If you pass props/attributes to native HTML elements that do not exist in the HTML specification React will not render them. However, if you use a custom element (i.e., not a standard HTML element) then arbitrary/custom attributes will be added to custom elements (e.g., `<x-my-component custom-attribute="foo"/>`).
- The `class` attribute has to be written `className`
- The `for` attribute has to be written `htmlFor`
- The `style` attribute takes a reference to an object of camel-cased style properties
- HTML form elements (e.g., `<input></input>` , `<textarea></textarea>` , etc.) created as React nodes support a few attributes/props that are affected by user interaction. These are: `value` , `checked` , and `selected` .
- React offers the `key` , `ref` , and `dangerouslySetInnerHTML` attributes/props that don't exist in DOM and take on a unique role/function.
- All attributes are camel-cased (e.g., `accept-charset` is written as `acceptCharset`) which differs from how they are written in HTML.
- The following are the HTML attributes that React supports (shown in camel-case):

```
accept acceptCharset accessKey action allowFullScreen allowTransparency alt
async autoComplete autoFocus autoPlay capture cellPadding cellSpacing challenge
charSet checked classID className colSpan cols content contentEditable
contextMenu controls coords crossOrigin data dateTime default defer dir
disabled download draggable encType form formAction formEncType formMethod
formNoValidate formTarget frameBorder headers height hidden high href hrefLang
htmlFor httpEquiv icon id inputMode integrity is keyParams keyType kind label
lang list loop low manifest marginHeight marginWidth max maxLength media
mediaGroup method min minLength multiple muted name noValidate nonce open
optimum pattern placeholder poster preload radioGroup readOnly rel required
reversed role rowSpan rows sandbox scope scoped scrolling seamless selected
shape size sizes span spellCheck src srcDoc srcLang srcSet start step style
summary tabIndex target title type useMap value width wmode wrap
```

Inlining CSS on Element Nodes

To apply inline CSS to a React node you have to pass a `style` attribute/prop with an object value containing CSS properties and values.

For example, in the code below I am passing a `style` prop referencing (i.e., `inlineStyle`) an object containing CSS properties and values:

```
var inlineStyles = {backgroundColor: 'red', fontSize: 20};

var reactNodeLi =  React.createElement('div',{style: inlineStyles}, 'styled')

ReactDOM.render(reactNodeLi, document.getElementById('app1'));
```

The resulting HTML will look like so:

```
<div id="app1">
  <div style="background-color: red;font-size: 20px;" data-reactid=".0">styled</div>
</div>
```

Note two things in the JavaScript code above:

1. I didn't have to add the "px" to the `fontSize` value because React did it for me.
2. When writing CSS properties in JavaScript you have to use the camelCased version of the CSS property (e.g., `backgroundColor` not `background-color`).

Notes

- Vendor prefixes other than ms should begin with a capital letter. This is why WebkitTransition has an uppercase "W".
- CamelCased CSS properties shouldn't be a surprise given this is how it is done when accessing properties on DOM nodes from JS (e.g., `document.body.style.backgroundImage`)
- When specifying a pixel value React will automatically append the string "px" for you after your number value except for the following properties:

```
columnCount fillOpacity flex flexGrow flexShrink fontWeight
lineClamp lineHeight opacity order orphans strokeOpacity
widows zIndex zoom
```

Using Built-in Element Factories

React provides built-in shortcuts for creating commonly used HTML element nodes. These shortcuts are called React element factories.

> A ReactElement-factory is simply a function that generates a ReactElement with a particular type property.

Using a built-in element factory (i.e., `React.DOM.li()`), the React element node `<li id="li1">one</li>` can be created like so:

```
// uses React.DOM.li(props, children);
var reactNodeLi = React.DOM.li({id: 'li1'}, 'one');
```

instead of using:

```
// uses React.createElement(type, prop, children)
var reactNodeLi = React.createElement('li', {id: 'li1'}, 'one');
```

Below I list out all of the built in node factories:

```
a,abbr,address,area,article,aside,audio,b,base,bdi,bdo,big,blockquote,body,br,button,
canvas,caption,cite,code,col,colgroup,data,datalist,dd,del,details,dfn,dialog,div,dl,
dt,em,embed,fieldset,figcaption,figure,footer,form,h1,h2,h3,h4,h5,h6,head,header,hgrou
p,
hr,html,i,iframe,img,input,ins,kbd,keygen,label,legend,li,link,main,map,mark,menu,
menuitem,meta,meter,nav,noscript,object,ol,optgroup,option,output,p,param,picture,
pre,progress,q,rp,rt,ruby,s,samp,script,section,select,small,source,span,strong,
style,sub,summary,sup,table,tbody,td,textarea,tfoot,th,thead,time,title,tr,track,
u,ul,var,video,wbr,circle,clipPath,defs,ellipse,g,image,line,linearGradient,mask,
path,pattern,polygon,polyline,radialGradient,rect,stop,svg,text,tspa
```

Notes

- If you are using JSX you might not ever use factories
- React has a built-in helper for you to create custom factories. It's `React.createFactory(type)`.

Defining React Node Events

Events can be added to React nodes much like events can be added to DOM nodes. In the code example below I am adding a very simple `click` and `mouseover` event to a React `<div>` node.

> source code

Note how the property name for an event in React starts with 'on' and is passed in the `props` argument object to the `ReactElement()` function.

React creates what it calls a `SyntheticEvent` for each event, which contains the details for the event. Similar to the details that are defined for DOM events. The `SyntheticEvent` instance, for an event, is passed into the events handlers/callback functions. In the code below I am logging the details of a SyntheticEvent instance.

> source code

Every SyntheticEvent object instance has the following properties.

```
boolean bubbles
boolean cancelable
DOMEventTarget currentTarget
boolean defaultPrevented
number eventPhase
boolean isTrusted
DOMEvent nativeEvent
void preventDefault()
boolean isDefaultPrevented()
void stopPropagation()
boolean isPropagationStopped()
DOMEventTarget target
number timeStamp
string type
```

Additional properties are available depending upon the type/category of event that is used. For example the `onClick` syntheticEvent event also contains the following properties.

```
boolean altKey
number button
number buttons
number clientX
number clientY
boolean ctrlKey
boolean getModifierState(key)
boolean metaKey
number pageX
number pageY
DOMEventTarget relatedTarget
number screenX
number screenY
boolean shiftKey
```

The table below outlines the unique SyntheticEvent properties for each type/category of events.

Event Type/Category:	Events:	Event Specific Properties:
Clipboard	onCopy, onCut, onPaste	DOMDataTransfer, clipboardData
Composition	onCompositionEnd, onCompositionStart, onCompositionUpdate	data
Keyboard	onKeyDown, onKeyPress, onKeyUp	altKey, charCode, ctrlKey, getModifierState(key), key, keyCode, locale, location, metaKey, repeat, shiftKey, which
Focus	onChange, onInput, onSubmit	DOMEventTarget, relatedTarget
Form	onFocus, onBlur	
Mouse	onClick, onContextMenu, onDoubleClick, onDrag, onDragEnd, onDragEnter, onDragExit onDragLeave, onDragOver, onDragStart, onDrop, onMouseDown, onMouseEnter, onMouseLeave onMouseMove, onMouseOut, onMouseOver, onMouseUp	altKey, button, buttons, clientX, clientY, ctrlKey, getModifierState(key), metaKey, pageX, pageY, DOMEventTarget relatedTarget, screenX, screenY, shiftKey,
Selection	onSelect	

Touch	onTouchCancel, onTouchEnd, onTouchMove, onTouchStart	altKey DOMTouchList changedTouches, ctrlKey, getModifierState(key), metaKey, shiftKey, DOMTouchList targetTouches, DOMTouchList touches,
UI	onScroll	detail, DOMAbstractView view
Wheel	onWheel	deltaMode, deltaX, deltaY, deltaZ,
Media	onAbort, onCanPlay, onCanPlayThrough, onDurationChange, onEmptied, onEncrypted, onEnded, onError, onLoadedData, onLoadedMetadata, onLoadStart, onPause, onPlay, onPlaying, onProgress, onRateChange, onSeeked, onSeeking, onStalled, onSuspend, onTimeUpdate, onVolumeChange, onWaiting	
Image	onLoad, onError	
Animation	onAnimationStart, onAnimationEnd, onAnimationIteration	animationName, pseudoElement, elapsedTime
Transition	onTransitionEnd	propertyName, pseudoElement, elapsedTime

Notes

- React normalizes events so that they behave consistently across different browsers.
- Events in React are triggered on the bubbling phase. To trigger an event on the capturing phase add the word "Capture" to the event name (e.g., `onClick` would become `onClickCapture`).
- If you need the browser event details you can access this by using the `nativeEvent` property found in the SyntheticEvent object passed into React event hander/callback.
- React doesn't actually attach events to the nodes themselves, it uses event delegation.
- `e.stopPropagation()` or `e.preventDefault()` should be triggered manually to stop event propagation instead of `return false;` .
- Not all DOM events are provided by React. But you can still make use of them using

React lifecycle methods.

JavaScript Syntax Extension (a.k.a, JSX)

In the previous chapter we learned how to create React nodes using plain ES5 JavaScript code. In this chapter we look at creating React nodes using the JSX syntax extension.

After this Chapter JSX will be used for the remainder of this book unless invoking a `React.createElement()` function is required for the sake of clarity.

What Is JSX?

JSX is an XML/HTML-like syntax used by React that extends ECMAScript so that XML/HTML-like text can co-exist with JavaScript/React code. The syntax is intended to be used by preprocessors (i.e., transpilers like Babel) to transform HTML-like text found in JavaScript files into standard JavaScript objects that a JavaScript engine will parse.

Basically, by using JSX you can write concise HTML/XML-like structures (e.g., DOM like tree structures) in the same file as you write JavaScript code, then Babel will transform these expressions into actual JavaScript code. Unlike the past, instead of putting JavaScript into HTML, JSX allows us to put HTML into JavaScript.

By using JSX one can write the following JSX/JavaScript code:

```
var nav = (
    <ul id="nav">
      <li><a href="#">Home</a></li>
      <li><a href="#">About</a></li>
      <li><a href="#">Clients</a></li>
      <li><a href="#">Contact Us</a></li>
    </ul>
);
```

And Babel will transform it into this:

```
var nav = React.createElement(
   "ul",
   { id: "nav" },
   React.createElement(
      "li",
      null,
      React.createElement(
         "a",
         { href: "#" },
         "Home"
      )
   ),
   React.createElement(
      "li",
      null,
      React.createElement(
         "a",
         { href: "#" },
         "About"
      )
   ),
   React.createElement(
      "li",
      null,
      React.createElement(
         "a",
         { href: "#" },
         "Clients"
      )
   ),
   React.createElement(
      "li",
      null,
      React.createElement(
         "a",
         { href: "#" },
         "Contact Us"
      )
   )
);
```

You can think of JSX as a shorthand for calling `React.createElement()`.

The idea of mixing HTML and JavaScript in the same file can be a rather contentious topic. Ignore the debate. Use it if you find it helpful. If not, write the React code required to create React nodes. Your choice. My opinion is that JSX provides a concise and familiar syntax for defining a tree structure with attributes that does not require learning a templating language or leaving JavaScript. Both of which can be a win when building large applications.

It should be obvious but JSX is easier to read and write over large pyramids of JavaScript function calls or object literals (e.g., contrast the two code samples in this section). Additionally the React team clearly believes JSX is better suited for defining UI's than a traditional templating (e.g., Handlebars) solution:

> markup and the code that generates it are intimately tied together. Additionally, display logic is often very complex and using template languages to express it becomes cumbersome. We've found that the best solution for this problem is to generate HTML and component trees directly from the JavaScript code such that you can use all of the expressive power of a real programming language to build UIs.

Notes

- Don't think of JSX as a template but instead as a special/alternative JS syntax that has to be compiled. I.e., JSX is simply converting XML-like markup into JavaScript.
- The Babel tool is a subjective selection from the React team for transforming ES* code and JSX syntax to ES5 code. You can learn more about Babel at http://babeljs.io/ or by reading the Babel handbook.
- The merits of JSX in four bullet points:
  - Less technical people can still understand and modify the required parts. CSS developers and designers will find JSX more familiar than JavaScript alone. I.e., HTML markup, using JSX, looks like HTML markup instead of a pyramid of `createElement()` function.
  - You can leverage the full power of JavaScript in HTML and avoid learning or using a templating language. JSX is not a templating solution. It is a declarative syntax used to express a tree structure of UI nodes and components.
  - By adding a JSX transformation step you'll find errors in your HTML you might otherwise miss.
  - JSX promotes the idea of inline styles. Which can be a good thing.
- Beware of JSX Gotchas.
- JSX is a separate thing from React itself. JSX does not attempt to comply with any XML or HTML specifications. JSX is designed as an ECMAScript feature and the similarity to XML/HTML is only at the surface (i.e., it looks like XML/HTML so you can just write something familiar). A JSX specification is currently being drafted to by used by anyone as an a XML-like syntax extension to ECMAScript without any defined semantics.
- In JSX, `<foo-bar />` alone is valid while `<foo-bar>` alone isn't. You have to close all tags, always.

Creating React Nodes With JSX

Working off the knowledge given in the previous chapter you should be familiar with creating React nodes using the `React.createElement()` function. For example, using this function one can create React nodes which both represent HTML DOM nodes and custom HTML DOM nodes. Below I use this familiar function to create two React nodes.

```
//React node, which represents an actual HTML DOM node
var HTMLLi = React.createElement('li', {className:'bar'}, 'foo');

//React node, which represents a custom HTML DOM node
var HTMLCustom = React.createElement('foo-bar', {className:'bar'}, 'foo');
```

To use JSX instead (assuming you have Babel setup) of `React.createElement()` to create these React nodes one just has to replace `React.createElement()` function calls with the HTML/XML like tags which represent the HTML you'd like the virtual DOM to spit out. The above code can be written in JSX like so.

```
//React node, which represents an actual HTML DOM node
var HTMLLi = <li className="bar">foo</li>;

//React node, which represents a custom HTML DOM node
var HTMLCustom = <foo-bar className="bar" >foo</foo-bar>;
```

Notice that the JSX is not in a JavaScript string format and can just be writing as if you are writing it inside of an `.html` file. As stated many times already JSX is then transformed back into the `React.createElement()` functions calls by Babel. You can see the transformation occurring in the following JSFiddle (i.e., Babel is transforming JSX to JavaScript, then React is creating DOM nodes).

> source code

If you were to examine the actual HTML produced in the above JSfiddle it would look like so:

```
<body>
    <div id="app1"><li class="bar" data-reactid=".0">foo</li></div>
    <div id="app2"><foo-bar class="bar" data-reactid=".1">foo</foo-bar></div>
</body>
```

Creating React nodes using JSX is as simple as writing HTML like code in your JavaScript files.

Notes

- JSX tags support the XML self close syntax so you can optionally leave the closing tag off when no child node is used.
- If you pass props/attributes to native HTML elements that do not exist in the HTML specification, React will not render them to the actual DOM. However, if you use a custom element (i.e., not a stand HTML element) then arbitrary/custom attributes will be added to custom elements (e.g., `<x-my-component custom-attribute="foo" />`).
- The `class` attribute has to be written `className`
- The `for` attribute has to be written `htmlFor`
- The `style` attribute takes an object of camel-cased style properties
- All attributes are camel-cased (e.g., `accept-charset` is written as `acceptCharset`)
- To represent HTML elements, ensure the HTML tag is lower-cased
- The following are the HTML attributes that React supports (shown in camel-case):

```
accept acceptCharset accessKey action allowFullScreen allowTransparency alt
async autoComplete autoFocus autoPlay capture cellPadding cellSpacing challenge
charSet checked classID className colSpan cols content contentEditable
contextMenu controls coords crossOrigin data dateTime default defer dir
disabled download draggable encType form formAction formEncType formMethod
formNoValidate formTarget frameBorder headers height hidden high href hrefLang
htmlFor httpEquiv icon id inputMode integrity is keyParams keyType kind label
lang list loop low manifest marginHeight marginWidth max maxLength media
mediaGroup method min minLength multiple muted name noValidate nonce open
optimum pattern placeholder poster preload radioGroup readOnly rel required
reversed role rowSpan rows sandbox scope scoped scrolling seamless selected
shape size sizes span spellCheck src srcDoc srcLang srcSet start step style
summary tabIndex target title type useMap value width wmode wrap
```

Rendering JSX to DOM

The `ReactDOM.render()` function can be used to render JSX expressions to the DOM. Actually, after Babel transforms the JSX all it is doing is rendering nodes created by `React.createElement()`. Again, JSX is just a stand in expression for having to write out the `React.createElement()` function calls.

In the code example I am rendering a `<li>` element and a custom `<foo-bar>` element to the DOM using JSX expressions.

> source code

Once rendered to the DOM, the HTML will look like so:

```
<body>
    <div id="app1"><li class="bar" data-reactid=".0">foo</li></div>
    <div id="app2"><foo-bar classname="bar" children="foo" data-reactid=".1">foo</foo-
bar></div>
</body>
```

Just remember that Babel is taking the JSX in your JavaScript files transforming it to React nodes (i.e., `React.createElement()` functions calls) then using these nodes created by React (i.e., the Virtual DOM) as a template for creating a real html DOM branch. The part where the React nodes are turned into the real DOM nodes and added to the DOM in an HTML page occurs when `ReactDOM.render()` is called.

Notes

- Any DOM nodes inside of the DOM element in which you are rendering will be removed/replaced.
- `ReactDOM.render()` does not modify the DOM element node in which you are rendering React.
- Rendering to an HTML DOM is only one option with React, other rendering APi's are available. For example, it is also possible to render to a string (i.e., `ReactDOMServer.renderToString()`) on the server-side.
- Re-rendering to the same DOM element will only update the current child nodes if a change (i.e., diff) has occurred or a new child node have been added.
- Don’t ever call `this.render()` yourself, leave that to React

Using JavaScript Expressions in JSX

Hopefully by now it's obvious that JSX is just syntactical sugar that gets converted to real JavaScript. But what happens when you want to intermingle real JavaScript code within JSX? To write a JavaScript expression within JSX you will have to surround the JavaScript code in `{ }` brackets.

In the React/JSX code below I am mixing JavaScript expressions (e.g., `2+2`), surround by `{ }` among the JSX that will eventually get evaluated by JavaScript.

> source code

The JSX transformation will result in the follow:

```
var label = '2 + 2';
var inputType = 'input';

var reactNode = React.createElement(
  'label',
  null,
  label,
  ' = ',
  React.createElement('input', { type: inputType, value: 2 + 2 })
);

ReactDOM.render(reactNode, document.getElementById('app1'));
```

Once this code is parsed by a JavaScript engine (i.e., a browser) the JavaScript expressions are evaluated and the resulting HTML will look like so:

```
<div id="app1">
    <label data-reactid=".0"><span data-reactid=".0.0">2 + 2</span><span data-reactid=
".0.1"> = </span><input type="input" value="4" data-reactid=".0.2"></label>
</div>
```

Nothing that complicated is going on here once you realize that the brackets basically escape the JSX. The `{ }` brackets simply tells JSX that the content is JavaScript so leave it alone so it can eventually be parsed by a JavaScript engine (e.g., `2+2`). Note that `{ }` brackets can be used anywhere among the JSX expressions as long as the result is valid JavaScript.

Notes

- If you have to escape brackets (i.e., you want brackets in a string) use `{'{}'}`.

Using JavaScript Comments in JSX

You can place JavaScript comments anywhere in React/JSX you want except locations where JSX might expect a React child node. In this situation you'll have to escape the comment using `{ }` so that JSX knows to pass that on as actual JavaScript.

Examine the code below, make sure you understand where you'll have to tell JSX to pass along a JS comment so a child React node is not created.

```
var reactNode = <div /*comment*/>{/* use {'{}'} here to comment*/}</div>;
```

In the above code if I had not wrapped the comment inside of the `<div>` with `{ }` brackets then Babel would have converted the comment to a React text node. The outcome, unintentionally, without the `{ }` would be:

```
var reactNode = React.createElement(
  "div",
  null,
  "/* use ",
  "{}",
  " here to comment*/"
);
```

Which would result in the following HTML that would have unintended text nodes.

```
<div data-reactid=".0">
    <span data-reactid=".0.0">/* use </span>
    <span data-reactid=".0.1">{}</span>
    <span data-reactid=".0.2"> here to comment*/</span>
</div>
```

Using Inline CSS in JSX

In order to define inline styles on React nodes you need to pass the `style` prop/attribute a JavaScript object or reference to an object containing CSS properties and values.

In the code below I first setup a JavaScript object, named `styles`, containing inline styles. Then I use the `{ }` brackets to reference the object that should be used for the value of the style prop/attribute (e.g., `style={styles}`).

> source code

Notice that, the CSS properties are in a camelCased form similar to what is used when writing CSS properties in JavaScript. This is required because JavaScript does not allow hyphens in names.

When the above React/JSX code is transformed by Babel, and then parsed by a JavaScript engine, the resulting HTML will be:

```
<div style="color:red;background-color:black;font-weight:bold;" data-reactid=".0">test
</div>
```

Notes

- Vendor prefixes other than ms should begin with a capital letter. This is why WebkitTransition has an uppercase "W".
- CamelCased CSS properties shouldn't be a suprise given this is how it is done when accessing properties on DOM nodes from JS (e.g., `document.body.style.backgroundImage`)
- When specifying a pixel value React will automatically append the string "px" for you after your number value except for the following properties:

```
columnCount fillOpacity flex flexGrow flexShrink fontWeight lineClamp lineHeight
opacity order orphans strokeOpacity widows zIndex zoom
```

Defining Attributes/Props in JSX

In the previous chapter, section 4.4, I discussed passing `React.createElement(type, props, children)` attributes/props when defining React nodes. Since JSX is transformed into `React.createElement()` function calls you basically already have a understanding of how React node attributes/props work. However, since JSX is used to express XML-like nodes that get turned into HTML, attribute/props are defined by adding the attributes/props to JSX expressions as name-value attributes.

In the code example below I am defining a React `<li>` element node, using JSX, with five attributes/props. One is a non-standard HTML attribute (e.g., `foo:'bar'`) and the others are known HTML attributes.

```
var styles = {backgroundColor:'red'};
var tested = true;
var text = 'text';

var reactNodeLi = <li id=""
                      data-test={tested?'test':'false'}
                      className="blue"
                      aria-test="test"
                      style={styles}
                      foo="bar">
                          {text}
                  </li>;

ReactDOM.render(reactNodeLi, document.getElementById('app1'));
```

The JSX when it is transformed will look like this (note that attributes/props just become arguments to a function):

```
var reactNodeLi = React.createElement(
    'li',
    { id: '',
        'data-test': tested ? 'test' : 'false',
        className: 'blue',
        'aria-test': 'test',
        style: styles,
        foo: 'bar' },
    text
);
```

When the `reactNodeLi` node is render to the DOM it will look like this:

```
<div id="app1">
    <li id="true"
        data-test="test"
        class="blue"
        aria-test="test"
        style="background-color:red;"
        data-reactid=".0">
            text
    </li>
</div>
```

You should note the following:

1. Leaving an attribute/prop blank results in that attribute value becoming true (e.g., `id=""` becomes `id="true"` and `test` becomes `test="true"`)
2. The `foo` attribute, because it was not a standard HTML attribute, doesn't become a final HTML attribute.

Notes

- If an attribute/prop is duplicated the last one defined wins.
- If you pass props/attributes to native HTML elements that do not exist in the HTML specification, React will not render them. However, if you use a custom element (i.e., not a standard HTML element) then arbitrary/custom attributes will be added to custom elements (e.g., `<x-my-component custom-attribute="foo" />`).
- Omitting the value of an attribute/prop causes JSX to treat it as true (i.e., `<input checked id type="checkbox" />` becomes `<input checked="true" id="true" type="checkbox">`). This will even occur for attributes/props that are not presented in the final HTML due to the fact they are not actual HTML attributes.
- The `class` attribute has to be written `className`
- The `for` attribute has to be written `htmlFor`
- The `style` attribute takes a reference to an object of camel-cased style properties
- HTML form elements (e.g., `<input></input>` , `<textarea></textarea>` , etc.) created as React nodes support a few attributes/props that are affected by user interaction. These are: `value` , `checked` , and `selected` .
- React offers the `key` , `ref` , and `dangerouslySetInnerHTML` attributes/props that don't exist in DOM and take on a unique role/function.
- All attributes are camel-cased (e.g., `accept-charset` is written as `acceptCharset`) which differs from how they are written in HTML.
- The following are the HTML attributes that React supports (shown in camel-case):

```
accept acceptCharset accessKey action allowFullScreen allowTransparency alt
async autoComplete autoFocus autoPlay capture cellPadding cellSpacing challenge
charSet checked classID className colSpan cols content contentEditable
contextMenu controls coords crossOrigin data dateTime default defer dir
disabled download draggable encType form formAction formEncType formMethod
formNoValidate formTarget frameBorder headers height hidden high href hrefLang
htmlFor httpEquiv icon id inputMode integrity is keyParams keyType kind label
lang list loop low manifest marginHeight marginWidth max maxLength media
mediaGroup method min minLength multiple muted name noValidate nonce open
optimum pattern placeholder poster preload radioGroup readOnly rel required
reversed role rowSpan rows sandbox scope scoped scrolling seamless selected
shape size sizes span spellCheck src srcDoc srcLang srcSet start step style
summary tabIndex target title type useMap value width wmode wrap
```

Defining Events in JSX

In the previous chapter, in section 4.7, it was explained and demonstrated how events are defined on React nodes. To do the same thing in JSX you add the same camelCased event and the corresponding handler/callback as a prop/attribute of the JSX representing the React node.

Below is the none JSX way of adding an event to a React node (example from Previous chapter, section 4.7):

> source code

The above code written using JSX:

> source code

Note that we are using the `{ }` brackets to connect a JS function to an event (i.e., `onMouseOver={mouseOverHandler}`). This style of adding events to nodes mimics the DOM 0 style of inlining events on HTML elements (Don't worry, just mimics, does not really create inline events in the DOM).

React supports the following events and event specific syntheticEvent properties:

Event Type/Category:	Events:	Event Specific Properties:
Clipboard	onCopy, onCut, onPaste	DOMDataTransfer, clipboardData
Composition	onCompositionEnd, onCompositionStart, onCompositionUpdate	data
Keyboard	onKeyDown, onKeyPress, onKeyUp	altKey, charCode, ctrlKey, getModifierState(key), key, keyCode, locale, location, metaKey, repeat, shiftKey, which
Focus	onChange, onInput, onSubmit	DOMEventTarget, relatedTarget
Form	onFocus, onBlur	
	onClick, onContextMenu, onDoubleClick, onDrag, onDragEnd, onDragEnter,	altKey, button, buttons, clientX, clientY, ctrlKey,

Mouse	onDragExit onDragLeave, onDragOver, onDragStart, onDrop, onMouseDown, onMouseEnter, onMouseLeave onMouseMove, onMouseOut, onMouseOver, onMouseUp	getModifierState(key), metaKey, pageX, pageY, DOMEventTarget relatedTarget, screenX, screenY, shiftKey,
Selection	onSelect	
Touch	onTouchCancel, onTouchEnd, onTouchMove, onTouchStart	altKey DOMTouchList changedTouches, ctrlKey, getModifierState(key), metaKey, shiftKey, DOMTouchList targetTouches, DOMTouchList touches,
UI	onScroll	detail, DOMAbstractView view
Wheel	onWheel	deltaMode, deltaX, deltaY, deltaZ,
Media	onAbort, onCanPlay, onCanPlayThrough, onDurationChange, onEmptied, onEncrypted, onEnded, onError, onLoadedData, onLoadedMetadata, onLoadStart, onPause, onPlay, onPlaying, onProgress, onRateChange, onSeeked, onSeeking, onStalled, onSuspend, onTimeUpdate, onVolumeChange, onWaiting	
Image	onLoad, onError	
Animation	onAnimationStart, onAnimationEnd, onAnimationIteration	animationName, pseudoElement, elapsedTime
Transition	onTransitionEnd	propertyName, pseudoElement, elapsedTime

Notes

- React normalizes events so that they behave consistently across different browsers.
- Events in React are triggered on the bubbling phase. To trigger an event on the capturing phase add the word "Capture" to the event name (e.g., `onClick` would

become `onClickCapture`).

- If you need the browser event details for a given event you can access this by using the `nativeEvent` property found in the SyntheticEvent object passed into React event hander/callbacks.
- React does not actually attach events to the nodes themselves, it uses event delegation.
- `e.stopPropagation()` or `e.preventDefault()` should be triggered manually to stop event propagation instead of `returning false;` .
- Not all DOM events are provided by React. But you can still make use of them using React lifecycle methods.

Basic React Components

This chapter will show how React nodes are used to create basic React components.

What Is a React Component?

The next section will provide a mental model around the nature of a React component and cover details around creating React components.

Typically a single view of a user interface (e.g., the tree or trunk) is divided up into logical chunks (e.g., branches). The tree becomes the starting component (e.g., a layout component) and then each chunk in the UI will become a sub-component that can be divided further into sub components (i.e., sub-branches). This not only keeps the UI organized but it also allows data and state changes to logically flow from the tree, to branches, then sub branches.

If this description of React components is cryptic then I would suggest that you examine any application interface and mentally start dividing the UI into logical chunks. Those chunks potentially are components. React components are the programatic abstraction (i.e., UI, events/interactions, state changes, DOM changes) making it possible to literally create these chunks and sub-chunks. For example, a lot of application UI's will have a layout component as the top component in a UI view. This component will contain several sub-components, like maybe, a search component or a menu component. The search component can then be divided further into sub-components. Maybe the search input is a separate component from the button that invokes the search. As you can see, a UI can quickly become a tree of components. Today, software UI's are typically created by crafting a tree of very simple single responsibility components. React provides the means to create these components via the `React.createClass()` function (or, `React.Component` if using ES6 classes). The `React.createClass()` function takes in a configuration object and returns a React component instance.

A React component is basically any part of a UI that can contain React nodes (via `React.createElement()` or JSX). I spent a lot of time up front talking about React nodes so that the ingredients of a React component would be firmly understood. Sounds simple until one realizes that React components can contained other React sub-components which can result in a complex tree of components. This is not unlike the idea that React nodes can contain other React nodes in a Virtual DOM. It might hurt your brain, but if you think hard about it all a component does it wraps itself around a logical set of branches from a tree of nodes. In this sense, you define an entire UI from components using React but the result is a tree of React nodes that can easily be translated to something like an HTML document (i.e., tree of DOM nodes that produces a UI).

Creating React components

A React component that will potentially contain `state` can be created by calling the `React.createClass()` function. This function takes one argument object used to specify the details of the component. The available component configuration options are listed below (a.k.a., component specifications).

`render()`	A required value, typically a function that returns React nodes, other React components, or `null` / `false`
`getInitialState()`	Function which returns an object containing initial value of `this.state`
`getDefaultProps()`	Function which returns an object containing values to be set on `this.props`
`propTypes`	Object containing validation specifications for props
`mixins`	Array of mixins (object containing methods) that can be share among components
`statics`	Object containing static methods
`displayName`	String, naming the component, used in debugging messages. If using JSX this is set automatically.
`componentWillMount()`	Callback function invoked once immediately before the initial rendering occurs
`componentDidMount()`	Callback function invoked immediately after the initial rendering occurs
`componentWillReceiveProps()`	Callback function invoked when a component is receiving new props
`shouldComponentUpdate()`	Callback function invoked before rendering when new props or state are being received
`componentWillUpdate()`	Callback function invoked immediately before rendering when new props or state are being received.
`componentDidUpdate()`	Callback function invoked immediately after the component's updates are flushed to the DOM
`componentWillUnmount()`	Callback function invoked immediately before a component is unmounted from the DOM

The most important component configuration option is `render`. This configuration option is required and is a function that returns React nodes and components. All other component configurations are optional.

The following code is an example of creating a `Timer` React component from React nodes using `React.createClass()`.

Make sure you read the comments in the code.

> source code

It looks like a lot of code. However, the bulk of the code simply involves creating a `<Timer/>` component and then passing the `createClass()` function creating the component a configuration object containing 5 properties (`getInitialState` , `tick` , `componentDidMount` , `componentWillUnmount` , `render`).

Notice that `Timer` is capitalized. When creating custom React components you need to capitalize the name of the component. Additionally, the value `this` among the configuration options refers to the component instance created. We'll discuss the component API in more detail at the end of this chapter. For now, just meditate on the configuration options available when defining a React component and how a reference to the component is achieved using the `this` keyword. Also note, that in the code example above I added my own custom instance method (i.e., `tick`) during the creation of the `<Timer/>` component.

Once a component is mounted (i.e., created) you can use the component API. The api contains four methods.

API method	Example	Description
`setState()`	`this.setState({mykey: 'my new value'});` `this.setState(function(previousState, currentProps) { return {myInteger: previousState.myInteger + 1}; });`	Primary method used to re-render a component and sub components.
`replaceState()`	`this.replceState({mykey: 'my new value'});`	Like `setState()` but does not merge old state just deletes it uses new object sent.
`forceUpdate()`	`this.forceUpdate(function(){//callback});`	Calling `forceUpdate()` will cause `render()` to be called on the component, skipping `shouldComponentUpdate()` .
`isMounted()`	`this.isMounted()`	isMounted() returns true if the component is rendered into the DOM, false otherwise.

The most commonly used component API method is `setState()` . Its usage will be covered in the React Component State chapter.

Notes

- The component callback configuration options (`componentWillUnmount` , `componentDidUpdate` , `componentWillUpdate` , `shouldComponentUpdate` , `componentWillReceiveProps` , `componentDidMount` , `componentWillMount`) are also known as "lifecycle methods" because these various methods are executed at specific points in a component's life.
- The `React.createClass()` function is a convenience function that creates component instances for you (via JavaScript `new` keyword).
- The `render()` function should be a pure function. Which means:

> it does not modify component state, it returns the same result each time it's invoked, and it does not read from or write to the DOM or otherwise interact with the browser (e.g., by using `setTimeout`). If you need to interact with the browser, perform your work in `componentDidMount()` or the other lifecycle methods instead. Keeping `render()` pure makes server rendering more practical and makes components easier to think about.

Components Return One Node/Component

The `render` configuration value defined when creating a component should return only one React node (or, component). This one node/component can contain any number of children. In the code below the `<reactNode>` is the starting node. Inside of this node any number of sibling and child nodes can be returned.

> source code

Note that if you want to return React nodes on more than one line (taking advantage of whitespace) you will have to surround the returned value in `()`. In the code below the JSX defining (i.e., rendered) the `MyComponent` is returned in `()`.

> source code

An error will occur if more than one starting React node is attempted to be returned. If you think about it, the error occurs because returning two `React.createElement()` functions isn't possible with JavaScript.

> source code

Thus, the above code will result in the following error:

```
babel.js:62789 Uncaught SyntaxError: embedded: Adjacent JSX elements must be wrapped i
n an enclosing tag (10:3)
   8 |     return (
   9 |             <span>test</span>
> 10 |             <span>test</span>
     |    ^
  11 |     );
  12 |   }
  13 | });
```

Commonly, developers will add a wrapping element `<div>` element to avoid this error.

This issue also concerns components. Just like React nodes, if you are returning a component, only one starting component can be returned but that component can have unlimited children.

> source code

If you return two sibling components, the same error will occur.

> source code

```
VM7370 babel.js:62789 Uncaught SyntaxError: embedded: Adjacent JSX elements must be wr
apped in an enclosing tag (10:2)
   8 |     return (
   9 |         <MyChildComponent/>
> 10 |         <AnotherChildComponent/>
     |   ^
  11 |     );
  12 |   }
  13 | });
```

Referring to a Component Instance

When a component is `render` 'ed', a React component instance is created from the passed configuration options. One can gain access to this instance and it's properties (e.g., `this.props`) and methods (e.g., `this.setState()`) in two ways.

The first way is by using the `this` keyword from within a configuration function option. In the code example below all of the `console.log(this)` statements will refer to the component instance.

> source code

The other way to gain a reference to a component instance involves making use of the return value from calling `ReactDOM.render()` . In other words, the `ReactDOM.render()` function will return a reference to the top most rendered component.

> source code

Notes

- The `this` keyword will commonly be used from within a component to access instance properties like `this.props.[NAME OF PROP]` , `this.props.children` , and `this.state` ,. It will also be used to call class methods/properties, that all components share like `this.setState` and `this.replaceState()` .

Defining Events on Components

Events can be added to React nodes inside of a components `render` configuration option (discussed in Ch. 4 section 4.7 and Ch. 5 section 5.8).

In the code example, two React events (i.e., `onClick` & `onMouseOver`) are set on React nodes (via JSX) using what looks like component props.

> source code

Basically, React when rendering a component looks for special React prop events (e.g., `onClick`) and treats these props differently from other props (all React events shown in table below). Obviously the difference being, that eventing in the real DOM is being wired up behind the scenes.

Part of this wiring is auto binding the context of the handler/callback to the scope of the component instance. In the code example below the value of `this` inside of the handlers/callbacks will reference the component instance itself.

> source code

React supports the following events and event specific syntheticEvent properties:

Event Type/Category:	Events:	Event Specific Properties:
Clipboard	onCopy, onCut, onPaste	DOMDataTransfer, clipboardData
Composition	onCompositionEnd, onCompositionStart, onCompositionUpdate	data
Keyboard	onKeyDown, onKeyPress, onKeyUp	altKey, charCode, ctrlKey, getModifierState(key), key, keyCode, locale, location, metaKey, repeat, shiftKey, which
Focus	onChange, onInput, onSubmit	DOMEventTarget, relatedTarget
Form	onFocus, onBlur	
	onClick, onContextMenu, onDoubleClick, onDrag, onDragEnd, onDragEnter,	altKey, button, buttons, clientX, clientY, ctrlKey,

Mouse	onDragExit onDragLeave, onDragOver, onDragStart, onDrop, onMouseDown, onMouseEnter, onMouseLeave onMouseMove, onMouseOut, onMouseOver, onMouseUp	getModifierState(key), metaKey, pageX, pageY, DOMEventTarget relatedTarget, screenX, screenY, shiftKey,
Selection	onSelect	
Touch	onTouchCancel, onTouchEnd, onTouchMove, onTouchStart	altKey DOMTouchList changedTouches, ctrlKey, getModifierState(key), metaKey, shiftKey, DOMTouchList targetTouches, DOMTouchList touches,
UI	onScroll	detail, DOMAbstractView view
Wheel	onWheel	deltaMode, deltaX, deltaY, deltaZ,
Media	onAbort, onCanPlay, onCanPlayThrough, onDurationChange, onEmptied, onEncrypted, onEnded, onError, onLoadedData, onLoadedMetadata, onLoadStart, onPause, onPlay, onPlaying, onProgress, onRateChange, onSeeked, onSeeking, onStalled, onSuspend, onTimeUpdate, onVolumeChange, onWaiting	
Image	onLoad, onError	
Animation	onAnimationStart, onAnimationEnd, onAnimationIteration	animationName, pseudoElement, elapsedTime
Transition	onTransitionEnd	propertyName, pseudoElement, elapsedTime

Notes

- React normalizes events so that they behave consistently across different browsers.
- Events in React are triggered on the bubbling phase. To trigger an event on the capturing phase add the word "Capture" to the end of the event name (e.g., `onClick`

would become `onClickCapture`).

- If you need the browser event details for a given event you can access this by using the `nativeEvent` property found in the SyntheticEvent object passed into React event hander/callbacks.
- React does not actually attach events to the nodes themselves, it uses event delegation.
- `e.stopPropagation()` or `e.preventDefault()` should be triggered manually to stop event propagation instead of `returning false;` .
- Not all DOM events are provided by React. But you can still make use of them using React lifecycle methods.

Composing Components

If it is not obvious React components can make use of other React components. That is, when defining a component the `render` configuration function can contain references to other components. When a component contains another component it can be said that the parent component owns the child component (aka composition).

In the code below the `BadgeList` component contains/owns the `BadgeBill` and `BadgeTom` component.

> source code

This code was purposefully simplistic in order to demonstrate component composition. In the next chapter, we will look at how the code will typically be written making use of props to create a generic `Badge` component. The generic Badge component can take any name value v.s. creating a unique badge and hard coding the name in the component.

Notes

- A key tenet of maintainable UI are composable components. React components by design are meant to contain other React components.
- Notice how HTML and previously defined components are mixed together in the `render()` configuration function.

Grokking Component Lifecycle's

React components live certain life events that are called lifecycle events. These lifecycle's events are tied to lifecycle methods. I discussed several of these methods at the start of this chapter when discussing the creation of components.

The lifecycle methods provide hooks into the phases and the nature of a component. In the code example, taken from section 6.2, I am console logging the occurrence of the lifecycle events `componentDidMount` , `componentWillUnmount` , and `getInitialState` lifecycle methods.

> source code

The methods can be divided into three categories (Mounting, Updating, and Unmounting phases).

Below I show a table for each category and the containing lifecycle methods.

Mounting Phase (happens once in a components life):

> "The first phase of the React Component life cycle is the Birth/Mounting phase. This is where we start initialization of the Component. At this phase, the Component's props and state are defined and configured. The Component and all its children are mounted on to the Native UI Stack (DOM, UIView, etc.). Finally, we can do post-processing if required. The Birth/Mounting phase only occurs once." - React In-depth

Method	Description
`getInitialState()`	is invoked before a component is mounted. Stateful components should implement this and return the initial state data.
`componentWillMount()`	is invoked immediately before mounting occurs.
`componentDidMount()`	is invoked immediately after mounting occurs. Initialization that requires DOM nodes should go here.

Updating Phase (happens over and over in a components life):

> "The next phase of the life cycle is the Growth/Update phase. In this phase, we get new props, change state, handle user interactions and communicate with the component hierarchy. This is where we spend most of our time in the Component's life. Unlike Birth or Death, we repeat this phase over and over." - React In-depth

Method	Description
`componentWillReceiveProps(object nextProps)`	is invoked when a mounted component receives new props. This method should be used to compare this.props and nextProps to perform state transitions using this.setState().
`shouldComponentUpdate(object nextProps, object nextState)`	is invoked when a component decides whether any changes warrant an update to the DOM. Implement this as an optimization to compare this.props with nextProps and this.state with nextState and return false if React should skip updating.
`componentWillUpdate(object nextProps, object nextState)`	is invoked immediately before updating occurs. You cannot call this.setState() here.
`componentDidUpdate(object prevProps, object prevState)`	is invoked immediately after updating occurs.

Unmounting Phase (happens once in a components life):

> "The final phase of the life cycle is the Death/Unmount phase. This phase occurs when a component instance is unmounted from the Native UI. This can occur when the user navigates away, the UI page changes, a component is hidden (like a drawer), etc. Death occurs once and readies the Component for Garbage Collection." - React In-depth

Method	Description
`componentWillUnmount()`	is invoked immediately before a component is unmounted and destroyed. Cleanup should go here.

Notes

- `componentDidMount` and `componentDidUpdate` are good places to put other libraries' logic.
- Read React In-depth for a detailed look into the React component lifecycle events
- Mounting Phase follows this order:
    1. Initialize / Construction
    2. `getDefaultProps()` *(React.createClass)* or `MyComponent.defaultProps` *(ES6 class)*

3. `getInitialState()` *(React.createClass)* or `this.state = ...` *(ES6 constructor)*
4. `componentWillMount()`
5. `render()`
6. Children initialization & life cycle kickoff
7. `componentDidMount()`

- Updating Phase follows this order:
  1. `componentWillReceiveProps()`
  2. `shouldComponentUpdate()`
  3. `componentWillUpdate()`
  4. `render()`
  5. Children Life cycle methods
  6. `componentDidUpdate()`
- Unmount Phase follows this order:
  1. `componentWillUnmount()`
  2. Children Life cycle methods
  3. Instance destroyed for Garbage Collection

Accessing Children Components/Nodes

If a component, when used, contains child React components or React nodes inside of the component (e.g., `<Parent><Child /></Parent>` or `<Parent><span>test<span></Parent>`) these React node or component instances can be accessed by using `this.props.children`.

In the code below the `Parent` component when used, contains two `<div>` React node children, which contains React text nodes. All of the children instances, inside of the component are accessible using `this.props.children`. In the code below I access these children inside of the `Parent` `componentDidMount` lifecycle function.

> source code

The `this.props.children` of the `Parent` component instance returns an array containing the references to the child React node instances. This array is logged out to the console. Additionally, in the `Parent` component I am logging out the child of the first `<div>` (i.e., a text node).

Note how when I use the `Parent2` component inside of the `Parent` component the child React node of `Parent2` only contains one `<span>` React node (i.e., `<span>child2text</span>`). Thus, `this.props.children` used in `componentDidMount` lifecycle function for `Parent2` component will be a direct reference to this `<span>` React node (i.e., not an array containing a single reference).

Given that `this.props.children` can potentially contain a wide set of nodes and components React provides a set of utilities to deal with this data structure. These utilities are listed below.

Utilitie	Description
`React.Children.map(this.props.children, function(){})`	Invoke fn on every immediate child contained within children with this set to thisArg. If children is a keyed fragment or array it will be traversed: fn will never be passed the container objects. If children is null or undefined returns null or undefined rather than an array.
`React.Children.forEach(this.props.children, function(){})`	Like React.Children.map() but does not return an array.
`React.Children.count(this.props.children)`	Return the total number of components in children, equal to the number of times that a callback passed to map or forEach would be invoked.
`React.Children.only(this.props.children)`	Return the only child in children. Throws otherwise.
`React.Children.toArray(this.props.children)`	Return the children opaque data structure as a flat array with keys assigned to each child. Useful if you want to manipulate collections of children in your render methods, especially if you want to reorder or slice this.props.children before passing it down.

Notes

- When there is only a single child, `this.props.children` will be the single child component itself without the array wrapper. This saves an array allocation.
- It can be confusing at first to realize that `children` are not what a component returns, but instead what is contained inside of component anywhere (including in a `render`) it is instantiated.

Using `ref` Attribute

The `ref` attribute makes it possible to store a reference to a particular React element or component returned by the component `render()` configuration function. This can be valuable when you need a reference, from within a component, to some element or component contained within the `render()` function.

To make a reference, place the `ref` attribute with a function value on any React element or component. Then, inside of the function, the first parameter within the scope of the function will be a reference to the element or component the `ref` is on.

For example in the code below I am console logging the reference out for each `ref`.

> source code

Notice (see console) that references to components return component instances while references to React elements return a reference to the actual DOM element in the HTML DOM (i.e., not a reference to the virtual DOM, but the actual HTML DOM).

A common use for `ref`'s are to store a reference on the component instance. In the code below I use the `ref` callback function on the text `<input>` to store a reference on the component instance so other instance methods have access to the reference via `this` (i.e., `this.textInput.focus()`).

> source code

Notes

- Refs can't be attached to a stateless function because the component does not have a backing instance.
- You might see a `ref` attribute with a string instead of a function; this is called a string `ref` and will likely be deprecated in the future. Function `ref`s are preferred.
- The `ref` callback function is called immediately after the component is mounted.
- References to a component instance allow one to call custom methods on the component if any are exposed when defining it.
- Writing refs with inline function expressions means React will see a different function object on every update, `ref` will be called with null immediately before it's called with the component instance. I.e., when the referenced component is unmounted and whenever the `ref` changes, the old `ref` will be called with `null` as an argument.
- React makes two suggestions when using refs: "Refs are a great way to send a message to a particular child instance in a way that would be inconvenient to do via

streaming Reactive props and state. They should, however, not be your go-to abstraction for flowing data through your application. By default, use the Reactive data flow and save refs for use cases that are inherently non-reactive." and "If you have not programmed several apps with React, your first inclination is usually going to be to try to use refs to "make things happen" in your app. If this is the case, take a moment and think more critically about where state should be owned in the component hierarchy. Often, it becomes clear that the proper place to "own" that state is at a higher level in the hierarchy. Placing the state there often eliminates any desire to use refs to "make things happen" – instead, the data flow will usually accomplish your goal."

Re-rendering A Component

You likely realize that calling `ReactDom.render()` is the initial kicking off of the rendering of a component and all sub components.

After the initial mounting of components, a re-rendering will occur when:

1. A component's `setState()` method is called
2. A component's `forceUpdate()` method is called

Anytime a component is re-rendered (or initially rendered) all of its children components are rendered inside of the virtual DOM possibly causing a change to the real DOM (i.e., the UI). The distinction I am making here is that just because a component is re-rendered in the virtual DOM, it does not always follow that an update to the DOM will occur.

In the code example below `ReactDOM.render(< App / >, app);` starts the first cascade of rendering, rendering `<App />` and `<Timer/>`. The next re-render occurs when the `setInterval()` calls the `setState()` component method, which causes `<App />` and `<Timer/>` to be re-rendered. Note the UI changes when the `now` state is changed.

> source code

The next re-render occurs when the `setTimeout()` is invoked and `this.forceUpdate()` is called. Note that simply updating the state (i.e., `this.state.now = 'foo';`) does not cause a re-render. I set the state using `this.state`, and then I have to call `this.forceUpdate()` so the component will re-render with the new state.

NEVER UPDATE THE STATE USING `this.state.`, DID IT HERE TO SHOW USE OF `this.forceUpdate()`.

React Component Properties

This section will discuss component properties (a.k.a., props).

What Are Component Props?

The simplest way to explain component props would be to say that they function similarly to HTML attributes. In other words, props provide configuration values for the component. For example, in the code below a `Badge` component is created and it is expecting a 'name' prop to be sent when the component is instantiated.

> source code

Inside the render function of the `<BadgeList>` component, where `<Badge>` is used, the `name` prop is added to the `<Badge>` component much like an HTML attribute is added to an HTML element (i.e., `<Badge name="Bill" />`). The `name` prop is then used by the `Badge` component (i.e., `this.props.name`) as the text node for the React `<div>` node rendered by the `Badge` component. This is similar to how an `<input>` can take a value attribute which it uses to display a value.

Another way to think about component props is that they are the configuration values sent to a component. If you look at the non-JSX version of the previous code example it should be obvious component props are just an object that gets passed to the `createElement()` function (i.e., `React.createElement(Badge, { name: "Bill" })`).

```
var Badge = React.createClass({
    displayName: "Badge",

    render: function render() {
        return React.createElement(
            "div",
            null, //no props defined, so null
            this.props.name //use passed this.prop.name as text node
        );
    }
});

var BadgeList = React.createClass({
    displayName: "BadgeList",

    render: function render() {
        return React.createElement(
            "div",
            null,
            React.createElement(Badge, { name: "Bill" }),
            React.createElement(Badge, { name: "Tom" })
        );
    }
});

ReactDOM.render(React.createElement(BadgeList, null), document.getElementById('app'));
```

This is similar to how props can be set directly on React nodes (see 4.4 and 5.7). However, when the `createElement()` function is passed a component definition (i.e., `Badge`) instead of a node, the props become available on the component itself (i.e., `this.props.name`). Component props make it possible to re-use the `<Badge>` component with any name.

In the previous code example examined in this section, the `BadgeList` component uses two `Badge` components each with their own `this.props` object. We can verify this by console logging out the value of `this.props` when a `Badge` component is instantiated.

> source code

Basically every React component instance has a unique instance property called `props` that starts as an empty JavaScript object. The empty object can get filled, by a parent component, with any JavaScript value/ reference. These values are then used by the component or passed on to child components.

Notes

- In ES5 environments/engines you won't be able to mutate `this.props` because its frozen (i.e., `Object.isFrozen(this.props) === true;`).

- You should consider `this.props` to be readonly.

Sending Component Props

Sending properties to a component entails adding HTML attribute like named values to the component when it is used, not when it is defined. For example, the below `Badge` component is defined first. Then, to send a prop to the `Badge` component, `name="Bill"` is added to the component when it is used (i.e., when `<Badge name="Bill" />` is rendered).

```
var Badge = React.createClass({
    render: function() {
        return <div>{this.props.name}</div>;
    }
});

ReactDOM.render(<Badge name="Bill" />, document.getElementById('app'));
```

Keep in mind anywhere a component is used a property can be sent to it. For example, the code from the previous section demonstrates the use of the `Badge` component and `name` property from within the `BadgeList` component.

```
var Badge = React.createClass({
    render: function() {
        return <div>{this.props.name}</div>;
    }
});

var BadgeList = React.createClass({
    render: function() {
        return (<div>
            <Badge name="Bill" />
            <Badge name="Tom" />
        </div>);
    }
});

ReactDOM.render(<BadgeList />, document.getElementById('app'));
```

Notes

- A components properties should be considered immutable and components should not internally alter the properties sent to them from above. If you need to alter the properties of a component then a re-render should occur; don't set `props` by adding/updating them using `this.props.[PROP] = [NEW PROP]` .

Getting Component Props

As discussed in section 6.4 a component instance can be accessed from any configuration option that uses a function by way of the `this` keyword. For example, in the code below the `this` keyword is used to access the `Badge` `props` from the component `render` configuration option (i.e., `this.props.name`).

```
var Badge = React.createClass({
    render: function() {
        return <div>{this.props.name}</div>;
    }
});

ReactDOM.render(<Badge name="Bill" />, document.getElementById('app'));
```

Nothing that difficult to grasp is happening if you look at the transformed JavaScript (i.e., JSX to JS)

```
var Badge = React.createClass({
    displayName: "Badge",

    render: function render() {
        return React.createElement(
            "div",
            null,
            this.props.name
        );
    }
});

ReactDOM.render(React.createElement(Badge, { name: "Bill" }), document.getElementById(
'app'));
```

The `{ name: "Bill" }` object is sent to the `createElement()` function along with a reference to the `Badge` component. The value `{ name: "Bill" }` is set as an instance property value of the component accessible from the `props` property (ie. `this.props.name === "Bill"`).

Notes

- You should consider `this.props` to be readonly, don't set props using `this.props.PROP = 'foo'` .

Setting Default Component Props

Default props can be set when a component is being defined by using the `getDefaultProps` configuration value.

In the code example below, the `Badge` component has a default value for the `name` prop.

> source code

Default props will be set on `this.props` if no prop is sent into the component. You can verify this by the fact that the `Badge` component instance with no `name` prop uses the default name `'John Doe'` .

Notes

- The `getDefaultProps` is invoked once and cached when the component is created.
- The `getDefaultProps` is run before any instances are created thus using `this.props` inside of the `getDefaultProps` will not work.
- Any objects returned by `getDefaultProps()` will be shared across instances, not copied.

Component Props More Than Strings

Before looking at validating props one needs to make sure they understand that a component prop can be any valid JavaScript value.

In the code example below I setup several default props containing several different JavaScript values.

> source code

Note how the `propArray` and `propObject` were overwritten with new values when the `MyComponent` instance is created.

The main take away here is that you are not limited to string values when passing prop values.

Validating Component Props

The enforcement of a prop can be validated when component instances are created.

When defining a component, the `propTypes` configuration option can be used to identify if and how props should be validated. In the code example below I'm checking to see that `propArray` and `propFunc` are in fact the correct data types and are sent into the component when it is instantiated.

> source code

I am not sending the correct props as specified using `propTypes` to demonstrate that doing so will cause an error. The above code will result in the following error showing up in the console.

```
Warning: Failed propType: Invalid prop `propArray` of type `object` supplied to `MyCom
ponent`, expected `array`

Warning: Failed propType: Required prop `propFunc` was not specified in `MyComponent`.

Uncaught TypeError: this.props.propFunc is not a function
```

Note however, the commented below will not cause an error:

```
ReactDOM.render(<MyComponent propArray={[1,2]} propFunc={function(){return 3;}} />, do
cument.getElementById('app'));
```

React offers several built in validators (e.g., `React.PropTypes[VALIDATOR]`) which I outline below (Note that creating custom validators are possible as well.):

Basic type validators:

These validators check to see if the prop is a specific JS primitive. By default all these are optional. In other words, the validation only occurs if the prop is set.

`React.PropTypes.string`	If prop is used, verify it is a string
`React.PropTypes.bool`	If prop is used, verify it is a boolean
`React.PropTypes.func`	If prop is used, verify it is a function
`React.PropTypes.number`	If prop is used, verify it is a number
`React.PropTypes.object`	If prop is used, verify it is a object
`React.PropTypes.array`	If prop is used, verify it is a array
`React.PropTypes.any`	If prop is used, verify it is of any type

Required type validators:

`React.PropTypes.[TYPE].isRequired`	Chaining the `.isRequired` on to any type validation to make sure the prop is provided (e.g., `propTypes:{propFunc:React.PropTypes.func.isRequired}`)

Element validators:

`React.PropTypes.element`	Is a React element.
`React.PropTypes.node`	Is anything that can be rendered: numbers, strings, elements or an array (or fragment) containing these types.

Enumerables validators:

`React.PropTypes.oneOf(['Mon','Fri'])`	Is one of several types of specific values.
`React.PropTypes.oneOfType([React.PropTypes.string,React.PropTypes.number])`	Is an object that could be one of many types

Array and Object validators:

`React.PropTypes.arrayOf(React.PropTypes.number),`	Is an array containing only one type of values.
`React.PropTypes.objectOf(React.PropTypes.number)`	Is an object containing only one type of property values
`React.PropTypes.instanceOf(People)`	Is object instance of specific constructor(uses `instanceof`)
`React.PropTypes.shape({color:React.PropTypes.string,size: React.PropTypes.number})`	Is object containing properties having a specific type

Custom validators:

`function(props, propName, componentName){}`	Supply your own function. For example: `propTypes: { customProp: function(props, propName, componentName) { if (!/matchme/.test(props[propName])) { return new Error('Validation failed!'); } } }`

```
propTypes: {
  customProp: function(props, propName, componentName) {
    if (!/matchme/.test(props[propName])) {
      return new Error('Validation failed!');
    }
  }
}
```

React Component State

This section will discuss component state.

What Is Component State?

Most components should simply take in props and render. But, components also offer state, and it is used to store information/data about the component that can change over time. Typically the change comes as a result of user events or system events (i.e., as a response to user input, a server request, or the passage of time).

According to the React documentation state should:

> Contain data that a component's event handlers may change to trigger a UI update. In real apps this data tends to be very small and JSON-serializable. When building a stateful component, think about the minimal possible representation of its state, and only store those properties in this.state. Inside of render() simply compute any other information you need based on this state. You'll find that thinking about and writing applications in this way tends to lead to the most correct application, since adding redundant or computed values to state means that you need to explicitly keep them in sync rather than rely on React computing them for you.

Working with Component State

Working with component state typically involves setting a components default state, accessing the current state, and updating the state.

In the code example below I am creating a `<MoodComponent />` that demonstrates the use of `getInitialState`, `this.state.[STATE]`, and `this.setState()`. If you click on the component in a web browser (i.e., the face) then it will cycle through the states (i.e., moods) available. Thus, the component has three potential states, tied to the UI, based on clicks from the UI user. Go ahead and click on the face in the results tab below.

> source code

Note that the `<MoodComponent />` has an initial state of ':|', that is set using `getInitialState: function() {return {mood: ':|'};}`, which is used in the component when it is first rendered by writing, `{this.state.mood}`.

To change the state, an event listener is added; in this case a click event (i.e., `onClick`) on the `<span>` node that will call the `changeMood` function. Within this function I use `this.setState()` to cycle to the next mood based on the current mood/state. After the state is update (i.e., `setState()` merges the changes) the component will re-render itself and the UI will change.

Things to keep in mind about React component state:

1. If a component has state, a default state should be provided using `getInitialState()`
2. State changes are typically how you start the re-rendering of a component and all sub components (i.e., children, grandchildren, great grand chidlren, etc.).
3. The only way a component should have its state update should be by using `this.setState()`. While other ways are possible (i.e. `forceUpdate()`), they should likely not be used (except maybe when integrating with third-party solutions).
4. You inform a component of a state change by using `this.setState()` to set a new state. This will result in re-render of the component and all children components that need re-rendered.
5. A state change merges new data with old data that is already contained in the state. But this is only a shallow update/merge, it won't do a deep update/merge.
6. A state change internally deals with calling re-renders. You should never have to call `this.render()` directly.
7. The state object should only contain the minimal amount of data needed for the UI. Don't place computed data, other React components, or props in the state object.

State vs. Props

A components `state` and `props` do share some common ground:

1. Both are plain JS objects
2. Both can have default values
3. Both should be accessed/read via `this.props` or `this.state`, but neither should be given values this way. I.e., both are readonly when using `this.`

However they are used for different reasons and in different ways.

Props:

1. Props are passed into a component from above, either a parent component or from the starting scope where React is originally rendered.
2. Props are intended as configuration values passed into the component. Think of them like arguments passed into a function (If you don't use JSX that is exactly what they are).
3. Props are immutable to the component receiving them. I.e., don't change props passed to a component from within the component

State:

1. State is a serializable representation of data (a JS object) at one point in time that typically is tied to UI
2. State should always start with a default value and then the state is mutated internally by the component using `setState()`
3. State can only be mutated by the component that contains the state. It is private in this sense.
4. Don't mutate the state of child components. A component should never have shared mutable state.
5. State should only contain the minimal amount of data needed to represent your UI's state. It should not contain computed data, other React components, or duplicated data from props.
6. State should be avoided if at all possible. I.e., stateless components are ideal, stateful components add complexity. The React documentation suggest: "A common pattern is to create several stateless components that just render data, and have a stateful component above them in the hierarchy that passes its state to its children via props. The stateful component encapsulates all of the interaction logic, while the stateless components take care of rendering data in a declarative way."

Creating Stateless Function Components

When a component is purely a result of `props` alone, no `state`, the component can be written as a pure function avoiding the need to create a React component instance. In the code example below `MyComponent` is the result of a function that returns the results from `React.createElement()`.

> source code

Having look at the same code not using JSX should clarify what is going on.

```
var MyComponent = function MyComponent(props) {
    return React.createElement(
        "div",
        null,
        "Hello ",
        props.name
    );
};

ReactDOM.render(React.createElement(MyComponent, { name: "doug" }), app);
```

Constructing a React component without calling `React.createClass()` is typically referred to as a stateless function component.

Stateless function components can't be passed component options (i.e., `render`, `componentWillUnmount`, etc.). However `.propTypes` and `.defaultProps` can be set as properties on the function.

The code example below demonstrates a stateless function component making use of `.propTypes` and `.defaultProps`.

> source code

Notes

- Make as many of your components as possible, as stateless components.